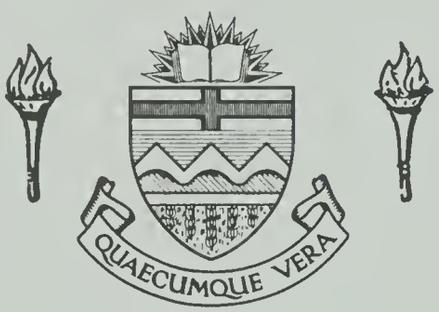

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR Desikan Jagannathan

TITLE OF THESIS Synchronization and Recovery in Distributed Databases

DEGREE FOR WHICH THESIS WAS PRESENTED Master of Science

YEAR THIS DEGREE GRANTED Fall 1983

THE UNIVERSITY OF ALBERTA

Synchronization and Recovery in Distributed Databases

by

Desikan Jagannathan

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE

OF Master of Science

Department of Computing Science

EDMONTON, ALBERTA

Fall 1983

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled Synchronization and Recovery in Distributed Databases submitted by Desikan Jagannathan in partial fulfilment of the requirements for the degree of Master of Science.

Abstract

A model for transaction management in a distributed database system is proposed. The basis for the model stems from the fact that the synchronization atomicity of a transaction can be derived from the atomicity of its subtransactions, and commit protocols. Consequently, the distributed concurrency control problem may be shown to reduce to three subproblems - local concurrency control at the nodes, distributed deadlock detection and multiple copy update. These subproblems are solved independently. A distributed database system is built by adding two software layers to local database systems, which are assumed to exist at the nodes.

Hierarchical and loosely coupled systems are the two extensions of the proposed model. They are characterized by the commit protocol used for transaction termination. A hierarchical interconnection relaxes the assumption of full connectivity between component databases. A loosely coupled system allows the user to view a collection of databases as a single database, even if the component systems do not participate in distributed transaction management.

Acknowledgements

I wish to thank Dr. Tony Marsland who, with an ideal combination of encouragement, criticism and independence, guided me during the course of this research. His many comments and suggestions contributed greatly to the contents and style of this thesis.

I am grateful to the members of my examining committee, Dr. W. Armstrong, Dr. D. Fenna and Dr. F. Chin, for their helpful comments.

I wish to thank the National Science and Engineering Research Council of Canada for their support of this research through the grant NSERC A7902.

I am indebted to the members of my family back home for their support and encouragement.

Finally, I wish to thank the authors of the game rogue, for all those hours of challenge.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Why Distribute Databases ?

Human organizations are inherently structured with multiple levels of control, authority and responsibility. However, Data Base Management Systems of the 60's and 70's have tended to centralize information about an enterprise because of cost and technological constraints. A centralized database has many advantages - it provides uniform and controlled access to shared data and enforces security and integrity constraints. A central authority, called the Database Administrator, is responsible for maintaining information needed by all the users. Recent advances in hardware technology have made available cheap processing power and storage, permitting information to be stored and processed close to the user. Consequently, a new technique in Systems Design, known as Distributed Systems, has emerged.

What can be qualified as a distributed system is a matter of definition. But the general consensus is that processors, data and control should be distributed in a distributed system. The model chosen is that a distributed system consists of a number computers, called nodes, connected together by a network. Each node has its own processor, memory and data and is largely autonomous. But nodes can cooperate among themselves to provide some

services to the user.

A Distributed Data Base (DDB) is defined as a set of component databases implemented at various nodes of the network. Each node has its own set of users who usually require access to data that is stored at that node. Occasionally, they request access to some data items which are located at different nodes. The DDB provides a uniform view of the component databases to all the users, irrespective of where they are located. The users see only a logical collection of data and they need not know the location of an object to read or write it. This is known as the location transparency of data. A Distributed Data Base Management System (DDBMS) is a set of programs implemented at various nodes of the network. The DDBMS is responsible for supporting and controlling user interactions with the DDB.

The objectives of distributed database systems are generally the same as those of centralized systems. However, some benefits are unique to distributed systems. Information can be kept close to where it is most needed, reducing communication costs and improving response times. Critical data can be duplicated, improving data availability. Distributed Databases are natural for applications in, for example, banking, airline reservation systems and government, which exhibit a well-defined hierarchy in their structure.

The algorithms used in the implementation of a DDBMS critically determine its performance. There are a number of problems like synchronization and recovery, to be addressed by the designer of a DDBMS. The solutions to these problems are more complex than those of centralized systems because of the parallelism and diversity inherent in distributed systems. In this report, some of these technical problems are addressed.

1.2 Architecture of DDBMS

Two well-defined and complementary architectures can be discerned in the implementation of a DDBMS. These are the Information and System Architectures [DER82]. The information architecture defines how the information is to be modeled, represented, allocated and accessed, while the system architecture defines processor characteristics, location, communications and protocols for transaction management [DER82]. This thesis concentrates on the system architecture, and in particular on transaction management.

1.3 Transactions

The concept of a transaction is well established in the database area [EGL76,Gra79,Gra81]. A transaction is some computation that reads the current state of a set of objects in the database and possibly changes the state of these objects. In essence, it consists of some *read* and *write* actions. A transaction can be expressed in a self-contained

query language or it can be embedded in a high-level language application program. Transactions originated by a number of users run concurrently on the system. One of the fundamental issues in designing a database system is to ensure that these transactions are isolated from each other and their effect is the same as when each of them is executed in some serial order. This is known as *serializability*, and the algorithms which enforce this property are known as concurrency control algorithms. A formal definition of serializability will be given later.

When a transaction requests access to a resource that is held by another, it is forced to wait until the resource becomes available again. Meanwhile, the second transaction itself may be waiting for another, and so on. If this *wait-for* relationship extends to a cycle, no transaction in the cycle will be able to proceed. This is known as the deadlock problem.

An *atomic* action is an action that appears to occur indivisibly in the system [Lam81]. *Read* and *Write* are examples of atomic actions and their atomicity is usually guaranteed by the operating system. A major objective of database systems is to guarantee the *atomicity* of transactions. This is done by ensuring that:

1. a partial execution of a transaction is not visible to other transactions
2. all the actions in a transaction are completed or none are, even when failures intervene [Lam81].

The first attribute is called the synchronization atomicity and it is enforced by the concurrency control algorithms. The second attribute is called the failure atomicity. In this presentation, the term "atomicity" is used to refer to both attributes. The term "synchronization atomicity" or "failure atomicity" is used to emphasize one attribute.

Recovery procedures in a database system are designed to enforce the failure atomicity of transactions. Typical failures that these procedures handle are the transaction and system failures.

The problems mentioned above, namely concurrency control, deadlock and recovery, are well understood in centralized systems. These solutions have been extended for distributed systems and several new ones proposed [BeG81,KKN83,Gra79]. These solutions are often very complex and only a few of them have been actually implemented. They are based on different assumptions about the system and very little is known about their comparative performance.

1.4 Distributed Transaction Management

A transaction in a distributed system may require access to data stored at different nodes. As a consequence, parts of it execute at different nodes. A transaction's initiation, migration and termination should be controlled by managers at the nodes where it is executed.

In a distributed system, nodes have only a partial knowledge of the system state and no single node knows the

complete system state. Transaction Managers (TMs), Data Managers (DMs) and Schedulers located at one or more nodes are the components of an architecture model for transaction management. TMs talk to schedulers and schedulers talk to DMs and also among themselves [BeG82]. Schedulers accept read/write requests from TMs and execute them in some order determined by the concurrency control algorithm used. The simplest architecture consists of only one scheduler located at a distinguished node and a TM and DM at every node. The scheduler controls all actions in the system and all TMs talk to it. Though this solution is easy to implement, it is not desirable for several reasons. Control is centralized and the entire system depends on one node, losing all benefits of distribution. Also, this node becomes the bottleneck for the entire system.

Another solution is to have a scheduler at each node, executing actions from different transactions in some order. In this scheme, it is assumed that all nodes are identically structured and they all use identical algorithms. Virtually all published algorithms assume one of the above models or some variations. For example, in timestamp-based algorithms all transactions are timestamped and actions in them are processed by different schedulers in the order of their timestamps.

In this report, a model is proposed for distributed transaction management in which individual transactions instead of individual actions are synchronized. Each node is

treated as a transaction processor, on top of which a distributed transaction processing layer is built. A transaction is split into a set of subtransactions which communicate among themselves. Each subtransaction is synchronized by a controller which is unaware of the existence of other nodes. It is proved that local concurrency control at the nodes, coupled with commitment protocols, ensure serializability. This is a bottom-up approach to design as opposed to the top-down approach, in which a DDBMS is designed first and components which must reside at different nodes are decided later. There are a number of benefits to the proposed approach:

* Heterogeneity in systems is allowed. Components can widely vary in their capabilities.
* Different nodes can use different concurrency control algorithms.
* Only minimal changes need be made to existing systems.
* Each node, by itself, can be a distributed database, permitting a tree-structured logical interconnection of DBMSs.

A disadvantage of this model is that it does not give priority to distributed transactions over local transactions. For example, a distributed transaction may be aborted and restarted in case of a deadlock, even though it may cost much less to abort a local transaction. Also, if a distributed system is designed from scratch,

application-dependent knowledge can be used to simplify design and improve performance.

Existing systems provide only a flat interconnection between component databases, in which every node has complete information about every other node. Such an interconnection is not suitable for very large networks. For example, in a national library network with hundreds of libraries and millions of books, it is unrealistic to assume that each node will have complete information about all other nodes. Hence, queries like "given a book, find the library" can not be supported easily.

As a first step in removing this restriction, the proposed transaction management model can be extended to a hierarchy of DDBMSs. In this model, each component database of a DDB can itself be a DDB, thus permitting a hierarchy. A transaction in this hierarchy consists of subtransactions, which consist of more subtransactions and so on. The execution tree of a transaction follows the topology of the interconnection. These are also called nested transactions, and the study of them is a currently active research area [Mos81,SpS83]. It will be shown in the next chapter that the proposed model can be extended to a special case of nested transactions, one in which no synchronization is necessary between descendents of a transaction.

There are often occasions when a user interacts with several databases, which themselves have nothing to do with each other. A sequence of interactions with these databases

is seen as a transaction by the user and each interaction is seen as a complete transaction by the databases concerned. This falls within the domain of Loosely Coupled Systems, a notion that is not supported by existing transaction models. The application of the proposed model to such systems will also be described in this thesis.

1.5 Organization of this Thesis

The characteristics of processors, communication and component databases are defined first. Subsequent discussions depend on these definitions. Existing models for transaction management will then be described. The new transaction management model will be presented after the motivations behind it are pointed out.

The chapters on Concurrency Control, Deadlock and Recovery are presented in the following format:

* Issues in Centralized Systems
* Issues in Distributed Systems
* Issues in the new model, with subsections on hierarchical DDBMS and loosely coupled systems.

The last chapter contains a summary of results obtained in this thesis and some suggestions for future work.

Chapter 2

Architecture Models

2.1 Design objectives

In the previous chapter, some benefits of distributed databases were pointed out. However, distributing data is purely a design issue. At the highest levels of abstraction, the user or the application programmer should see no difference between distributed and centralized databases. He should not have to worry about such details as the location of data or the failure of processors. At the user level, the system should exhibit the following properties:

a). *Location Transparency:* Although the data is geographically distributed and may move from place to place, the user can act as if all the data is at one node.

b). *Replication Transparency:* Although the same data item may be replicated at several nodes of the network, the programmer can treat them as if it were stored as a single item at a single site.

c). *Concurrency Transparency:* Although the system runs many transactions concurrently, to each transaction it appears as if it were the only activity in the system.

d). *Failure Transparency:* Either all the actions of a transaction occur or none of them occur. Once the transaction is committed, its effects survive all

hardware and software failures [TGG82]. Note that while this condition is impossible to achieve, it can be satisfied with a arbitrarily high probability.

Additionally, if the component databases are heterogeneous, the user should be given a globally consistent view of the data, independent of the data models used.

The Information (or Schema) Architecture defines how information is conceptually modeled, represented, allocated and accessed. The ANSI/SPARC 3-schema proposal for centralized database systems has been extended for distributed systems. It includes two additional levels - the Global Representation Schema and the Global Conceptual Schema [DER82,Dee81]. For heterogeneous systems supporting different data models, a common view of the data and a globally consistent access language are needed [Spa81]. Such an approach has been taken in Multibase, a heterogeneous distributed database system under development [SmB81]. Transactions in Multibase are expressed in ADAPLEX, an ADA-compatible query language. This thesis concentrates on the transaction management issues of synchronization, deadlock and recovery. Schema architecture does not directly affect these issues and hence will not be discussed further.

2.2 Distributed Systems

A distributed system is modeled by a set of nodes (computers) connected together by a network. Each node consists of a processor and two levels of memory - volatile and permanent [Mos81]. The contents of the volatile memory are assumed to be lost when the processor fails. Permanent memory is the disk storage and it is assumed to be ideal - the information stored in it is never corrupted and is always available to the processor. Techniques for constructing disk storage systems with arbitrarily low probabilities of failure are well known and a few such systems have actually been implemented [Ver77,Ver78,Ver79,Lam81].

Nodes of the distributed system do not have any shared memory and they communicate only by sending messages over the network. The network is viewed at a high level of abstraction and it is assumed to be constructed such that:

1. Messages sent from node A to node B arrive at node B within a finite time and in the order in which they were sent. Messages are not lost or duplicated [LeL79].
2. Two messages sent from nodes A and B to node C arrive at node C in an arbitrary order.
3. The failure of a node is reported by the network to any other node that tries to communicate with it. The network can not distinguish between the processor failure at a node and the failure of communications to that node.

This view is supported by a hierarchy of layers constituting the network. The status of an action completed by the distributed system is either *desired*, *undesired but anticipated* or *disaster* [Lam81]. The objective is to design a system such that it can handle any number of undesired but anticipated events like processor and communication failures. Disasters, like failure of all the nodes, are not handled.

2.3 Component Databases

In this presentation, each node is viewed as a transaction processor. Each node has a database and a transaction manager which accepts transactions that read and possibly modify the data. A database D is defined as a set of named entities:

$$D = \{ E_1, E_2, \ldots\ldots, E_n \}.$$

The entities may be such things as records, disk pages or files. The current contents of an entity, E_1 for example, is represented by value(E_1), while the current contents of the database, termed the database state [Ull82], is represented by state(D). The contents of an entity and the database vary with time, while the names of entities are invariant.

The values of some of these entities may be semantically related. For example, if seats-available, seats-reserved and capacity are entities in a database for airline reservations, they may be related by a constraint like:

seats-reserved + seats-available < 1.1 * capacity

Consistency constraints ensure that values of the database entities remain semantically related. We can associate a predicate with a consistency constraint such that the predicate is true if and only if the constraint is satisfied. The conjunction of predicates corresponding to various constraints on the database is known as the system predicate "I". The state of the database is consistent if I(state(D)) is true [Mos81]. The database has associated with it a transaction manager, TM. TM is a software module which accepts user transactions on the database and executes them. A transaction is modeled by a set of operations of the form:

read(E_i) return(value);

write(E_i,value);

The set of entities read by the transaction is known as its readset, R, and the set of entities written by the transaction is known as its writeset, W. A transaction either produces a new value for state(D) which satisfies I(state(D)) or leaves state(D) unchanged. The entities are stored in the permanent memory. The value stored in the permanent memory for an entity is termed its permanent copy. Transactions start with a *begin-transaction* command and end with an *end-transaction* command. When a *begin-transaction* command is issued, say for T, the TM creates a workspace for T in the volatile memory. When T issues a read command, a copy of the entity read (called a volatile copy) is made

available in T's workspace. When T issues a write command, the new value of the entity is written in its workspace. When T issues the *end-transaction* command, it goes through what is called the commitment phase [Gra79]. At the end of this phase all the changes made by T are reflected in the permanent copies of entities, and T's workspace is discarded. If the system crashes during the execution of T, the contents of the volatile memory are assumed to be lost. There are no checkpoints within T to partially restore it. Hence, T is aborted during restart. This model of transactions was proposed by Gray [Gra79] and Lampson [Lam81].

The read and write commands mentioned before are examples of actions, while a sequence of such actions is a transaction. For example, a transaction can be formed as follows:

```
begin-transaction
  x = read(E1)
  y = read(E2)
  write(E1, x + y )
end-transaction
```

The objective of database systems is to ensure the atomicity of transactions, given the atomicity of actions.

2.4 Model of DDB

A distributed database is defined as a set of component databases, $\{D_1, D_2 \ldots D_k\}$, such that each D_i, $i = 1,2 \ldots k$, along with its TM is implemented at node i. A DDBMS is a set of programs implemented at each node of the network to manage transactions that need access to data which is stored at various nodes.

Replication of data is a key issue in database design. It increases the availability of data items replicated and reduces communication costs for transactions that only read them. However, update transactions incur additional communication costs because of the synchronization that is needed to maintain the mutual consistency of the copies. Careful design is needed in balancing these conflicting requirements. Replicated data is identified by the $\equiv$ constraints.

Let each D_i be represented as $D_i = \{E_{1i}, E_{2i}, \ldots E_{mi}\}$. The current state of the DDB is represented by state(DDB). A fundamental property of distributed systems is that at any instant of time no single node knows the value of state(DDB) [LeL79].

Definition 1:

The $\equiv$ constraint on a pair of entities E_{mi}, E_{nj}, $i \neq j$, is satisfied if value(E_{mi}) = value(E_{nj}).

The $\equiv$ constraint identifies the multiple copies of the same data and strictly speaking they should be identical all the

time. Two entities which satisfy this constraint are treated as duplicates in a set union operation (that is, the constraint establishes an equivalence class).

Definition 2:

A distributed database is internally consistent if some invariant predicate I(state(DDB)) is true.

Definition 3:

A distributed database is externally consistent if all ≡ constraints on its entities are satisfied.

Two types of external consistencies, known as strong and weak external consistencies, must be distinguished:

a) A distributed database is said to satisfy strong external consistency if at any instance of time, all ≡ constraints on its entities are satisfied.

b) A distributed database is said to satisfy weak external consistency if, in the absence of any new activity, all ≡ constraints on its entities will be satisfied after an arbitrarily large amount of time [LeL81].

Maintaining strong external consistency involves heavy synchronization overheads. In a situation where all the users do not require the most current version of data, maintaining only a weak external consistency may be sufficient. For example, updating the library database once a day may be enough for most users. This is the basic idea behind the primary copy approach for multiple copy update (Chapter 3).

Definition 4:

A distributed database is said to be fully replicated if, $|D_i| = N$, $1 \le i \le k$, where N is some integer and for all m and j, $1 \le m \le N$, $2 \le j \le k$, there exists s, $1 \le s \le N$, such that $E_{m1} \equiv E_{sj}$.

In a fully replicated database the same set of entities is stored at all nodes.

Definition 5:

A distributed database is strictly partitioned if there are no $\equiv$ constraints to be satisfied on any pair of entities.

Semantically related data is stored at various nodes of the network and there is no duplication of data.

Definition 6:

A distributed system is said to be partitioned and partially replicated if it is not fully replicated and it is not strictly partitioned.

2.5 Distributed Transaction Management

As mentioned before, how a transaction is decomposed, executed and synchronized is a fundamental issue in DDBMS design. Some existing models of transaction management will be described first.

2.5.1 Existing Models

The model proposed by Bernstein and Goodman [BeG81,BeG82] is adapted to illustrate distributed

transaction processing in existing systems. The components of this model are Transaction Managers (TMs), Schedulers, Data Managers (DMs) and Buffer Managers (BMs). Transactions talk to TMs, TMs talk to schedulers and BMs and the schedulers talk to BMs and also to each other. A transaction consists of four types of commands, begin-transaction, end-transaction, read and write -- whose semantics are identical to those described before. A transaction issues all its commands to a single TM which routes them to different schedulers.

The centralized architecture [MeM78] for transaction processing is given in Figure 1. In this model, for a begin-transaction command from a transaction T, its TM negotiates work space for it at different BMs. Read and write actions in T are passed by its TM to the central scheduler. This scheduler sets a lock for the data item read or written and passes the command to the BM of the node where the data item is located. When T issues an end-transaction command, its TM passes it to the scheduler which coordinates the commitment of T and releases all locks associated with T.

The disadvantage of this approach is the centralized control, as the entire system depends on one node. This node becomes the limiting factor in the system performance. An improvement over this solution [MeM78,AlD76] employs backups of the scheduler at several nodes. TMs send requests to all schedulers, though only the primary scheduler is responsible

for control. When the primary fails, one of the backups becomes the new primary. Communication costs may be high in this scheme.

The decentralized model of transaction execution is given in Figure 2. In this model, every node has a TM, BM, DM and a scheduler. An action is sent by its TM to the scheduler of the node at which the data item resides. No single scheduler has complete knowledge about the states of transactions. Clearly, to satisfy the serializability property of transactions, some synchronization among them is necessary. In the following paragraph this problem is restated and some fundamental existing solutions are outlined.

TMs can be considered as producers of actions and schedulers their consumers [LeL79]. Actions are executed atomically by consumers. A transaction T is a set of actions which occur at different consumers. How can we make T atomic?

In the circulating sequencer solution to this problem [LeL81], producers organize themselves in a virtual ring and circulate a token through this ring. Only the producer which holds the token is allowed to initiate actions. It completes T and passes the token to the next producer. Producers are solely responsible for synchronization and consumers are passive. In timestamp-based methods [RSL78,Lam78], every transaction is assigned a timestamp by its TM. The consumers execute different actions in the order of their timestamps.

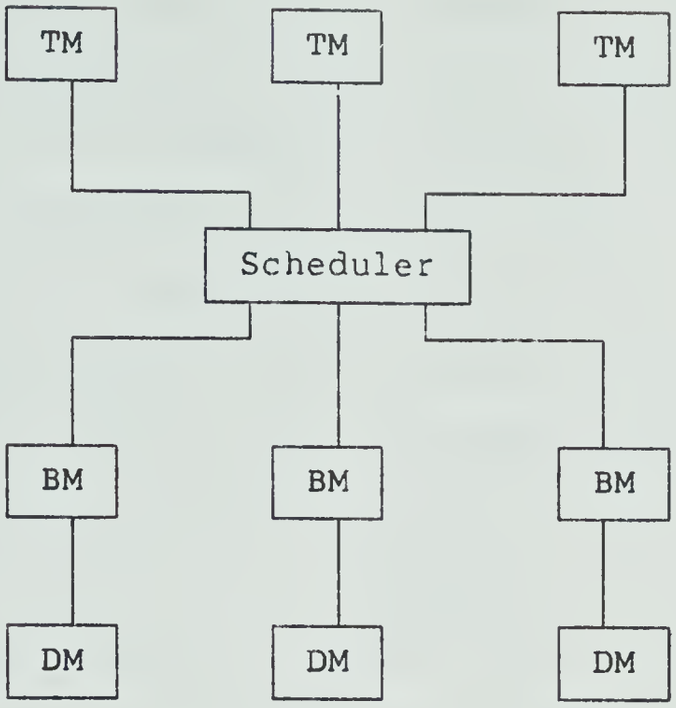

Figure 1. Centralized Architecture

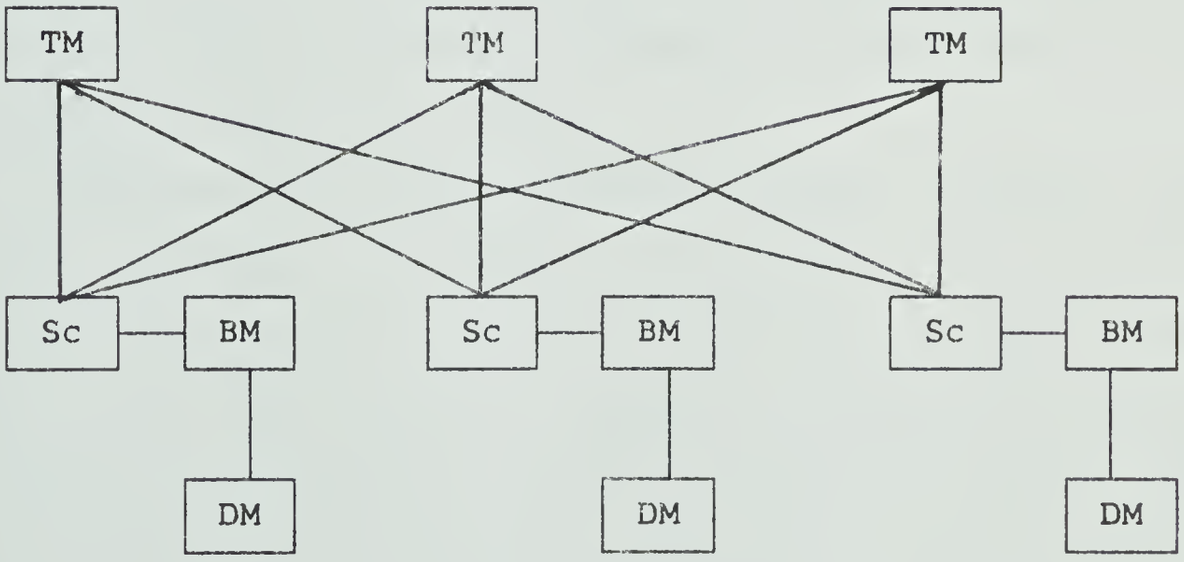

Figure 2. Distributed Architecture

In the two-phase locking solution [EGL76,TGG82], consumers guarantee mutual exclusion in access to objects between two time periods dynamically specified by the producers (through lock and unlock commands). A producer first locks all the entities accessed and then unlocks them.

A crucial observation in all these solutions is that individual actions are synchronized to guarantee atomicity of T. In my proposal it is individual transactions which are synchronized, based on the assumption that a consumer can guarantee atomicity of a set of actions. Actions in T are divided into subsets (called subtransactions) and each subtransaction is sent to one consumer. In the next chapter it is proved that this property, coupled with commit protocols, is a sufficient condition for serializability. An important feature of this solution is that each subtransaction at a consumer is considered a complete transaction and the consumer can synchronize it using any of the techniques mentioned above.

A DDBMS based on the existing techniques reflects the top-down approach to design. It consists of a set of identical modules which execute at different nodes. Each module knows the complete system structure. Only a flat interconnection is possible between component systems and they all have to be homogeneous. The model proposed removes these restrictions. It reflects the bottom-up approach to DDBMS design and only minimal capabilities are assumed of component systems. This makes it possible to integrate

heterogeneous systems. Heterogeneity can be not only in the data models, but also in concurrency control, deadlock and recovery algorithms.

2.6 Proposed Model

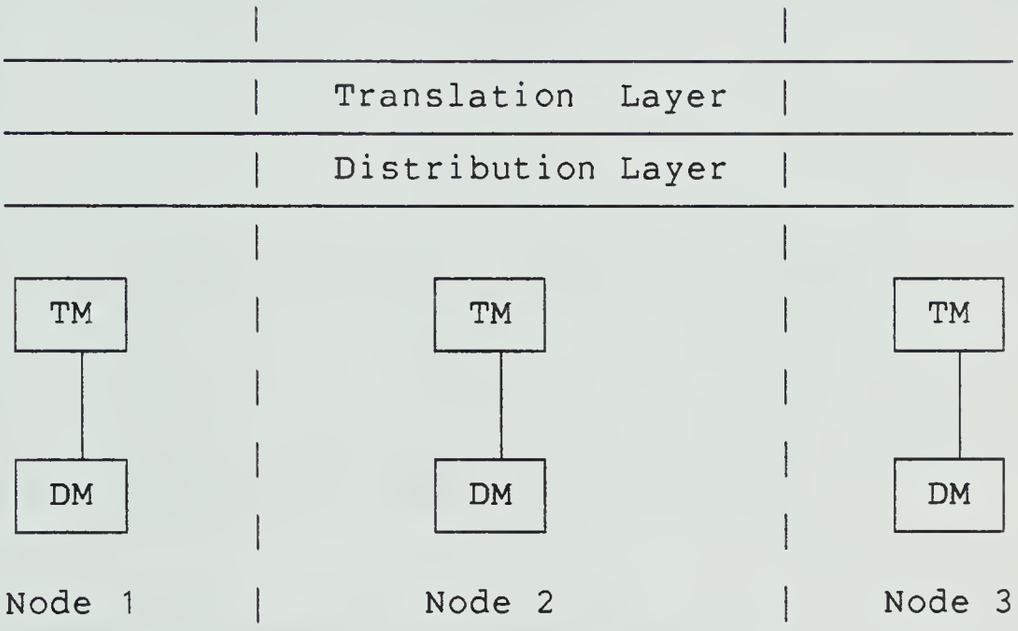

Figure 3. Proposed Model

In this model, each node is considered a database machine. It stores a set of entities and has a TM which accepts transactions for manipulating these entities. The DDBMS is built by adding two layers - the distribution layer and the translation layer - over each database machine. The distribution layer takes care of inter-nodal communication

and provides location transparency. The translation layer is responsible for the decomposition of user transactions. It also selects and executes protocols for multiple copy update. Above this layer, the user sees the database as if it is only at one node and without replication. His transactions are composed according to this view.

The distribution layer consists of a set of programs implemented at each node. These programs communicate among themselves for managing transactions. This layer is responsible for creating and deleting remote transactions, storing the states of transactions in cooperation with local TMs, supporting commit protocols, detecting global deadlocks and checkpointing. It provides a number of primitives for doing these operations. Some of them are discussed below and others will be described in subsequent chapters.

1. *begin-transaction*(node-id) *returns*(tid)

   This primitive is used to create subtransactions at various nodes. If begin-transaction is issued against the node at which it originated, it signals the beginning of a new global transaction. The transaction thus created is known as the coordinator. Its functions include coordination of the execution of subtransactions and consistent termination of the transaction (commit or abort).

2. *abort-transaction*(node-id,tid)

   Aborts the specified transaction at the node. All the

changes done by this transaction are undone at the node.

3. *secure-transaction*(node-id,tid) *returns*(status)

   This primitive is used to implement the two-phase commit protocol [Gra79]. It is invoked by the coordinator, after it has detected the completion of all subtransactions. The TM of each subtransaction is requested to secure the changes made by the subtransactions in disk storage (usually in a log file). If all the TMs send "secured" messages as replies, the coordinator can choose to commit the transaction.

4. *commit-transaction*(tid)

   This primitive is invoked by the coordinator and "commit" messages are sent to all subordinates.

2.6.1 Transaction Decomposition

A transaction is split into a set of subtransactions such that each of them executes entirely at one node. A transaction is allowed to make its requests dynamically. That is, its read and write sets need not be known in advance. Some communication is needed between subtransactions to perform operations across system boundaries. *send* and *receive* primitives are provided for this purpose. The semantics of these primitives is similar to that of remote procedure calls [Shr81,Lis79]. Send and receive operators are inserted within a transaction by the translation layer. Suppose that T_1 and T_2 are two subtransactions of T, executing at nodes 1 and 2

respectively. If T_1 wishes to perform an action at node 2, it sends a message of the format *proc-name(par1,par2...)* to T_2, where *proc-name* is the name of the procedure to be activated to perform the intended action(s). T_2 will receive this message and invoke the procedure *proc-name*. All the computations at node 2 are performed by T_2, and both T_1 and T_2 are within the scope of T. In existing models, a single TM would perform all the actions of T resulting in a centralized execution control of T. In the solution described above, execution of T is decentralized. For example, the procedure *proc-name* above can contain other remote actions.

The approach to transaction decomposition discussed above is similar to the dataflow solution for implementing distributed relational queries [Vin81]. In this solution, a transaction is represented by a query graph. The nodes of the graph are relational or transaction management operators. The edges reflect data dependencies between operators. Every edge in the graph which connects operators to be performed at different nodes is replaced by one *transmit* and one *receive* operator. Operations for each node are grouped together to form subtransactions.

The *send* primitive has the following syntax:

send(node-id,tid,msg) *returns*(status)

Three types of semantics are possible for the *send* primitive [Lis79]:

1. In the "no-wait" send, the sending process waits

only until the message is composed.

2. In the "synchronization" send, the sending process waits until the message is received by the target process.
3. In the "remote-invocation" send, the sending process waits for a response/result from the receiving process.

Any one of these semantics can be used.

Similarly, the *receive* primitive has the following syntax:

receive(node-id,tid,msg) *returns*(status)

The semantics of *receive* depends on the choice of *send*. For the "no-wait" send, the semantics of *receive* as defined by Dijkstra and Scholten [DiS80] can be used. They have assumed that each process has an associated message queue. Messages from other processes are appended to this queue and the receiving process retrieves them in the FIFO order.

Example:

Consider a distributed system with two nodes. The user's view of the database consists of two files, "supplier" and "supply". "supply" is stored entirely at node-2, whereas "supplier" is partitioned and one partition is stored at each node.

User's View:	supply(S#,P#)	
	supplier(S#,Sname).	
Distribution:	node-1	node-2
	supplier(S#,Sname)	supplier(S#,Sname)

supply(S#,P#)

Query (at node-1):

List all suppliers who supply "123NUTS".

Transaction:

```
 begin-transaction
    t = {x | (x,"123NUTS") ϵ supply)
    r = {x | ∃y: [(y,x) ϵ supplier] Λ [y ϵ t]}
 end-transaction
```

Decomposition:

```
 For node-1:
 begin-transaction
    t = {x | (x,"123NUTS") ϵ supply)
    r1 = {x | ∃y: [(y,x) ϵ supplier] Λ [y ϵ t]}
    tid = begin-transaction(node-2)
    send(node-2,tid,t)
    receive(r2)
    r = r1 union r2
 end-transaction

 For node-2:
 begin-transaction
    receive(t)
    r = {x | ∃y: [(y,x) ϵ supplier] Λ [y ϵ t]}
    send(node-1,tid,r)
 end-transaction
```

Several variations are possible in transaction decomposition to minimize such things as the communication traffic and the response time. This is known as the query optimization problem and is not dealt with in this thesis.

2.6.2 Hierarchical Interconnection

As mentioned before, the database machine at each node consists of a TM and a set of entities which it manipulates. The view of the DDBMS above the translation layer is identical to that of a single node database machine. This implies that each node can itself be a DDBMS. If this hierarchy is extended further, it results in a tree-structured interconnection of DDBMSs.

Why do we need this tree-structured interconnection? Current systems support only a flat interconnection in which each node knows the complete system structure and communicates with every other node. Control of the execution of a transaction is centered at a single TM, and it should have complete knowledge about the location of all objects accessed. If a hierarchical interconnection is permitted, the control of the execution of a transaction can be decentralized. Each node may need to know only about a few other nodes. This simplifies the task of maintaining system-wide directories.

Let $\{D_1, D_2 \ldots D_k\}$ be the set of component databases constituting the DDB. The set of entities seen by the user in the DDB is $\{D_1$ *union* $\ldots D_k\}$. In a hierarchical DDBMS,

any element of the set $\{D_1, D_2 \ldots D_k\}$ can by itself be a DDB. Assume that D_k is a distributed database while others are non-distributed. The set of elements known to D_k is the union of such sets known to its sons, and the transactions of the users at D_k operate only on entities in this set. We impose the following restriction on D_k:

> $\{D_1$ *union* $\ldots D_{k-1}\}$ and $\{D_k\}$ should be strictly partitioned.

If this restriction is not met, the following anomaly is possible:

> There are two entities, say E_1 and E_2, with an $\equiv$ constraint. One DDB knows only about E_1, while another DDB at a higher level in the hierarchy knows about both E_1 and E_2 and the $\equiv$ constraint between them.

The $\equiv$ constraint is meaningless in this example and such a situation is unacceptable. Any solution that removes this restriction should extend the tree model to a graph. The structure of a transaction in the graph model is complex, since some synchronization may be needed between its subtransactions. In this thesis we will not deal with this issue. No published soltion implements such a structure.

Transactions in a hierarchical DDBMS can be nested. For example, a subtransaction running at a node may create additional subtransactions at its sons without its parent knowing this fact. That is, a consumer can act as a producer to the next level of consumers. The execution tree of a

transaction follows the topology of DDBMS interconnection. Concurrency control for such systems will be described in Chapter 3.

The hierarchical structure reflects the logical interconnection of database systems and is independent of the structure of the underlying network.

Example:

The Newcastle Connection [BMR82] is the name of the software subsystem that has been added to UNIX, in order to connect them together. UNIX United is a distributed system based on the Newcastle Connection. It is composed of a (possibly large) set of interlinked standard UNIX systems, each one completely independent [BMR82]. These systems are arranged in a tree-structure and are addressed by the user in a format similar to the UNIX file structure. The proposed transaction model naturally fits in such an environment. Suppose that there is a procedure called *process-news* at each node; we can implement a transaction that sends some news to all systems in UNIX United as follows:

```
begin-transaction
    receive(proc-name,par1)
    invoke proc-name(par1)
end-transaction

process-news(newsfile) {
    write-news(newsfile)
    for each son do {
```

```
            t_i = begin-transaction(son)
            send(son,t_i,"process-news",newsfile)
        }
    }
```

There are some efforts under way to extend the transaction concept to non-database, general purpose distributed computing applications [Lis81,SpS83]. A group at the Carnegie Mellon University [SpS83] are investigating the idea of building a transaction kernel at every node to manage objects and modules (abstract data types). Their notion of transaction kernel is similar to the distribution layer in the proposed model. The object-oriented language CLU is being extended to serve as a distributed computing language [Lis82]. It supports transactions, which are allowed to manipulate remote objects. Nested transactions and message-based computations are fundamental to the design of such a language and a great deal of work still remains to be done.

2.6.3 Loosely Coupled Systems

There are often occasions when a user interacts with several databases which themselves have nothing to do with each other. Consider the following example, implementing a travel agent system:

A transaction consists of:

1. reservations with different airlines
2. car rental reservations

3. hotel reservations

4. credit card billing [Gra81].

Often, the user cancels reservations. It is inconceivable that the database systems of different airlines and hotels will cooperate for the convenience of the travel agent. Each of these systems treats portions of the transaction independently. Yet, it is seen as one complete transaction by the travel agent. This is an example of a transaction which has within its scope several others. An approach that is promising in this situation is the notion of *Compensating Transactions* (CT), suggested by Davies [Dav73]. Existing systems do not support such a notion. Gray [Gra81], while commenting on the limitations of existing transaction models, suggests that they be extended to support compensating transactions.

A CT is invoked to undo the effects of a committed transaction. Consider the following transaction T, initiated by an operator at a bank:

```
begin-transaction
  bal = read(acc#, acc-bal)
  write(acc#, acc-bal - 100)
end-transaction
```

Assume that T is completed. The operator detects later that the acc-bal should have been increased (deposit) and not decreased (withdrawal). This mistake can be corrected by running a credit-posting transaction, say T', as follows:

```
begin-transaction
  bal = read(acc#, acc-bal)
  write(acc#, acc-bal + 100)
end-transaction
```

We can consider T' as a CT for T. If some transaction had completed after T, but before T', it would have read the wrong balance for the account. This transaction should be undone and this may require some other transactions to be undone or aborted. This results in what is termed the domino effect [Ran75]. If T had in it an action like "dispense $10,000" (from an automatic teller), it cannot be undone and T can never be compensated. This is an example of a *real* action (not handled in the proposed model). A CT is allowed only if it does not force other completed transactions to abort. This is usually the responsibility of the user. However, in some special cases, we can build systems which automatically generate CTs.

Actions are *commutative* if their actual ordering is unimportant. For example, additive and subtractive actions on an object can be made commutative. Such actions do not make any value-based updates. An entity is said to be *commutable* (for want of a better adjective) if all the actions defined on it are commutative. A transaction is said to be *reversible* if its actions operate only on commutable entities. A CT is defined only for such a transaction.

As mentioned before, a transaction in the proposed model is decomposed into a number of subtransactions, one

for each node accessed. If in a particular node, say i, all the entities in the database are commutable, all subtransactions for D_i will be reversible. A CT can then be defined for each subtransaction at D_i.

Let DDB_1 be a distributed database and let D_i be a database that is loosely coupled to DDB_1, resulting in a new distributed database DDB_2. The users of DDB_2 see a database which is the union of entities in DDB_1 and D_i, and compose their transactions accordingly. However, since the entities in DDB_1 and D_i are semantically unrelated, a transaction is decomposed in such a way that all actions on D_i are grouped into a single subtransaction. Call this subtransaction c_s. c_s is processed at D_i and committed independently of its parent. Such a partial commitment does not violate system consistency constraints. There are consistency constraints on DDB_1 or D_i; but no single constraint involves entities in both. Let I_1, I_2 and I_i be the invariants on DDB_1, DDB_2 and D_i respectively.

Definition 7:

D_i at node *i* is loosely coupled to DDB_1 if:

1. there are no constraints involving entities in both DDB_1 and D_i, and I_2 can be expressed as
   $I_2(state(DDB_2)) = I_1(state(DDB_1)) \wedge I_i(state(D_i))$
2. all entities in D_i are commutable.

Note that this definition is asymmetric. Node *i* cannot treat DDB_1 as if it is loosely coupled.

Let T be a transaction initiated by a global user. Since the subtransaction of T at D_i is executed independently of its parent, some restrictions need to be imposed on the ordering of actions in T. These restrictions are defined based on the *collapsibility* of the precedence graph of T.

Let $T = \{ a_1, a_2, \ldots, a_n \}$, where each a_i, i=1,2...n, stands for an action in T.

Let $<_d$ be the partial ordering (precedence relation) on T. $a_1 <_d a_2$ implies that the action a_2 uses the value output by a_1 and that the action a_1 is to precede a_2.

Definition 8: (precedence graph) [CoD73]

The precedence graph G(W,U) of T is defined as follows:

$W = \{ a_i \mid a_i \in T \}$

$U = \{ (a_i, a_j) \mid (a_i <_d a_j) \wedge \neg(\exists k: a_i <_d a_k <_d a_j) \}$

Let $u = (a_x, a_y)$ be an element of U. Then, $tail(u) = a_y$ and $head(u) = a_x$.

Let $A = \{ a_j, a_{j+1} \ldots a_k \}$ be the set of all actions in T which operate on some entity in D_i.

Let G'(W',U') be a subgraph of G such that:

$W' = A$

$U' = \{ u \in U \mid tail(u) \in W' \wedge head(u) \in W' \}$

Collapsing of a subgraph of a given graph is the process in which the entire subgraph is replaced by a single new node. During this process, all edges entirely within the subgraph are discarded; all edges leading out of the subgraph are

replaced by edges leading out of the new node; all edges leading into the subgraph are replaced by edges leading into the new node.

Formally, collapsing of G' in G is the construction of a new graph $G^-(W^-,U^-)$ such that:

$$W^- = \{ a_i \mid (a_i \in W) \wedge \neg(a_i \in W') \} \textit{ union } \{ c \}$$

$$U^- = \{ (a_i,a_j) \in U \mid \neg(a_i \in W') \wedge \neg(a_j \in W') \}$$

union

$$\{ (a_i,c) \mid \neg(a_i \in W') \wedge \exists x \in W' : (a_i,x) \in U \}$$

union

$$\{ (c,a_i) \mid \neg(a_i \in W') \wedge \exists x \in W' : (x,a_i) \in U \}$$

[BeR81]

The subtransaction for D_i is collapsed into a single action c.

The property that should be satisfied by T is that G^- is acyclic. Otherwise, some communication is needed between the subtransaction that executes at D_i and its parent. In such a case, the action c cannot occur indivisibly. If D_i is loosely coupled to DDB_1, then no send or receive primitives are allowed within its subtransaction. Note that the subtransaction at D_i is committed independently of its parent. This property is immediately obvious from the collapsed graph as any coordinated commitment will create a simple two-node cycle.

As mentioned before, some minimal changes are necessary in database machines at nodes. Some primitives, like *secure-transaction*, must be supported by these systems at

the interface with the distribution layer. If D_i is loosely coupled to DDB_1, no such changes are necessary at node i. It participates only as a consumer.

At the first glance, it may seem as though the assumptions about entities in D_i are too restrictive and unrealistic. However, this is not the case. Two examples are given below to illustrate this point. Consider the travel agent case. In the absence of the extension suggested above, he will have to run 4 transactions corresponding to airline, hotel and car reservations and credit card billing. He is also responsible for undoing one or more of these transactions, if necessary. The notion of loosely coupled systems suggested in this thesis is aimed at removing this burden from the travel agent and delegating it to his system. The travel agent's view of the databases is suitably restricted to enable the system to take the responsiblity for the transaction. To the other users of the airline database, for example, it makes no difference whether the travel agent's transaction reserved a seat or his transaction's transaction reserved the seat.

The other, perhaps more important application of loosely coupled systems is in the extension of the transaction concept to non-database, general purpose distributed computing. The creation of temporary files at remote nodes is a typical operation in such an environment. This can be considered a commutable subtransaction initiated by a process in the distributed system. If the process is

aborted for some reason, all the temporary files created by it must be destroyed. The recovery procedures defined for a loosely coupled system will automatically do it for the user.

2.7 Summary

In this chapter, various components of a DDBMS architecture model are defined. Some existing models of transaction management are described and a new model is proposed. The new model is particularly suited for heterogeneous systems. Two special cases of the model -- hierarchical and loosely coupled systems are also described.

Chapter 3

Concurrency Control

3.1 Centralized Systems

Transactions are the basic unit of computation in a database system. Transactions initiated by several users may be run concurrently. The objective of concurrency control is to ensure that these transactions are effectively isolated from each other and that they get a consistent view of the database. Transactions should satisfy two fundamental properties - failure atomicity and serializability.

Failure atomicity requires that either all the actions in a transaction are completed or none are. Since failures during execution are possible, the system must have some facility to undo/redo actions (see Chapter 5). Even if there are no system failures, a transaction may be forced to abort because of incorrect computations (for example, a divide by 0) or invalid access requests.

3.1.1 Serializability

Let $\Pi = \{ T_1, T_2 \ldots, T_n \}$ be a set of transactions accessing the database. An execution of Π is serial if no two transactions run concurrently in it. Since every transaction is assumed to be a correct computation, serial execution of Π is correct. An execution of Π is *serializable* if it is computationally equivalent to some serial schedule. The transactions in Π are then said to satisfy

synchronization atomicity. The traditional example, used to illustrate concurrency control [Koh81] is to consider the case of 3 bank accounts A, B and C and three transactions T_1, T_2 and T_3. T_1 transfers \$100 from account A to account B, T_2 transfers \$50 from account B to account C, and T_3 prints the account balances of A, B and C. The consistency constraint is that the sum of account balances should be constant.

Initial account balances:

A = \$200

B = \$100

C = \$50

Transactions:

T_1:

```
    begin-transaction
1     A-bal = read(A)
2     B-bal = read(B)
3     write(A, A-bal - 100)
4     write(B, B-bal + 100)
    end-transaction
```

T_2:

```
    begin-transaction
1     B-bal = read(B)
2     C-bal = read(C)
3     write(B, B-bal - 50)
4     write(C, C-bal + 50)
    end-transaction
```

T_3:

```
        begin-transaction
1         A-bal = read(A)
2         B-bal = read(B)
3         C-bal = read(C)
        end-transaction
```

A schedule for Π in this example is an ordering of actions in T_1, T_2 and T_3. Suppose that the actions are executed in the following order:

.. steps 1-3 of T_1, 1-4 of T_2, 4 of T_1 ...

The resulting balance for B will be $200 when it should actually be $150. This is known as the lost update or ww-synchronization problem.

Suppose that in some other schedule, actions are processed in the following order:

.. steps 1-3 of T_1, 1-3 of T_3 and 4 of T_1 ..

T_3 will report account balances as $100, $100 and $50, when such a state never existed. T_3 will produce an output that can never again be repeated. This is known as the rw-synchronization problem.

Two actions are said to conflict if they both operate on the same data object and one of them is a write. Let the entity accessed by an action "a" be denoted by v(a). An action is of a certain type type(a) = x, $x \in \{r,w\}$, where r and w stand for read and write respectively.

Definition 1: (conflict)

Two actions a and b are in conflict, written a *conflict* b, if and only if

v(a) = v(b) Λ (type(a)=w | type(b)=w) [Sch78].

Definition 2: (schedule)

A schedule S = (A_s, <) consists of a set A_s of actions and a relation <, such that conflicting actions are ordered [Sch78].

a *conflict* b, a $\neq$ b, implies that a < b or b < a. a < b means that a precedes b in S. S establishes a partial ordering of all actions in Π.

Let S^+ be a serial schedule for Π. S is said to be serializable if

1. The final state of the database is identical to the state that would result if S^+ were the schedule.
2. Outputs of all transactions in Π are identical to outputs that would result if S^+ were the schedule.

Since we have assumed that each transaction, if run alone, would preserve the integrity of the database, a serializable schedule preserves the integrity of the database.

Partial ordering between actions introduces a dependency between transactions that contain them. This is characterized by the dependency graph of S.

Definition 3:

The dependency graph G(W,U) of a schedule is defined

as follows:

1. $W = \{ T_i \mid T_i \in T \}$
2. $U = \{ (T_i, T_j) \mid \exists a \in T_i, \exists b \in T_j : a < b \text{ is in } S \}$

[Sch78].

The nodes of the dependency graph are the transactions. If (T_1, T_2) is an edge in the graph it implies that T_1 read a value which was subsequently written by T_2 or T_1 wrote a value which was subsequently read or written by T_2.

Theorem 1:

A schedule is serializable if and only if its dependency graph does not contain a cycle [Sch78].

Note that serializability has been defined considering rw and ww conflicts together. It can also be defined separately for rw and ww conflicts [BeG81]. The serializability problem in this case is reduced to two subproblems, rw and ww synchronization, and different algorithms can be used to solve these subproblems.

3.1.2 Two-phase Locking

Two-phase locking is a technique used to detect conflicts. Before a transaction attempts to read or write an object, it obtains a lock for it of the corresponding type. Two transactions cannot be given a lock on an object if either one of them is a write. Transactions that obtain a proper lock on an object before reading or writing it are said to be well formed. A transaction is said to be two-phase if it obtains all the locks before unlocking any

one of them [EGL76]. Such a transaction exhibits two phases in its execution - the growing (locking) phase and the shrinking (unlocking) phase. It has been proved [EGL76] that if all transactions are well formed and two-phase, they are serializable. Two-phase locking is a sufficient condition for serializability.

The choice of an element for locking in a database, like a file or a record, determines the potential parallelism in the system and significantly affects performance [RiS77,RiS79]. The simple locking strategy just described can be extended to hierarchical locking [Gra79]. The hierarchy consists of the database, files, pages and records. A transaction can request locks at any level in the hierarchy. To detect conflicts within a hierarchy, an additional mode of locking, called the intention mode has been introduced [Gra79].

A major problem associated with locking is the possibility of deadlock. A deadlock occurs when a set of transactions can never be granted all the locks they need in order to reach completion. Solutions to deadlock in centralized systems are well known [CoD73].

3.2 Distributed Concurrency Control

A distributed concurrency control algorithm should address two problems - the internal consistency of the database and the mutual consistency of replicated copies. It has been shown [BeG81] that though a large number of such

algorithms have been published, they are all variations of two basic techniques, namely two-phase locking and timestamp ordering. These fundamental solutions are examined in some detail.

Two-Phase Locking:

Two-phase locking requires that all the actions in a transaction be processed by a single TM. Some of these actions may be remote. Before performing an operation on a remote object, the TM requests an appropriate lock from the object's scheduler (Figure 2). Note that there may be several schedulers in the distributed system. If TMs do not permit any parallelism within a transaction, the execution of transactions in a distributed system can be modeled as their execution in a centralized system [TGG82]. Hence, if all transactions in a distributed system are well formed and two-phase, they are serializable.

An implementation of this algorithm amounts to building a scheduler, a software module that receives lock and unlock requests and processes them according to the two-phase locking specification [BeG81]. Schedulers are distributed along with the database such that a data item and its scheduler reside at the same node.

A consequence of distributed two-phase locking is the possibility of global deadlocks. No single node will have sufficient information to detect such a deadlock. See Chapter 4 for further details.

Timestamp Ordering:

In this technique, each transaction is assigned a unique timestamp by its TM. Actions in it are given this timestamp before they are sent to a scheduler. The creation of timestamps requires loosely synchronized clocks at all the nodes. Timestamps are made unique systemwide by assigning a unique number to each node and appending this number to the lower order bits of the timestamp generated by it [Lam78].

The operation of a scheduler based on timestamp ordering is quite simple. Each scheduler resolves conflicts between actions in the order of their timestamps. For each data item in a node, its scheduler records the largest timestamps of read and write actions performed on it. The scheduler outputs actions received according to the following algorithm:

Let TS(a) be the timestamp associated with an action "a" operating on an object x.

```
if type(a) = r then
    if TS(a) < largest timestamp of any write on x
                    yet "accepted"
      then  reject a
      else
                    "accept" a and output it as soon as
                     all writes on x with smaller
                     timestamps are completed
else
```

```
if type(a) = w then
   if TS(a) < largest timestamp of any read
                or write on x yet "accepted"
   then reject a
   else
          "accept" a and output it as soon as
          all reads and writes on x with
          smaller timestamps are completed
          [BeG82].
```

An action is rejected if it tries to perform a read on an object which has a higher timestamp, since it may lead to inconsistencies. It can easily be proved that timestamp ordering always leads to serializable schedules [Lam78,RSL78,BeG82]. When an action is rejected, the transaction that tried to perform it is aborted (possibly at several sites) and restarted with a new and larger timestamp.

It has been shown [RSL78] that a transaction can find itself in a situation in which every time it restarts it is aborted subsequently, because of conflicts with other transactions. Such a transaction will go through infinite restart/abort cycles and can never be completed. A solution to this problem involves, among other approaches, assigning higher priority to restarted transactions. Several strategies have been proposed [RSL78] for coping with such problems.

A modified version of the algorithm just described, called conservative timestamp ordering [BeG81], avoids rejections by delaying operations instead. An action is delayed by the scheduler until it is sure that outputting it will cause no future operations to be rejected [BeG81]. This solution requires that each scheduler receive reads and writes from each TM in the order of their timestamps. The scheduler must have an operation from each TM in its input queue. It then scans this queue and chooses an action with the lowest timestamp. Since the earliest request made by each TM is known, outputting an operation in the queue with the lowest timestamp cannot lead to rejection of other actions in the future [BeG81]. Even if a TM has no actions for a scheduler, it periodically sends null operations with a timestamp. Note that this solution avoids rejections at the cost of excessive delays in processing actions.

Several variations of two-phase locking and timestamp ordering have been reported. Bernstein and Goodman have published a survey of these techniques [BeG81].

3.3 Concurrency Control in the Proposed Model

As described in Chapter 2, a transaction is decomposed into a set of subtransactions that are executed at different nodes. A subtransaction is executed only at one node. Subtransactions from a number of transactions run concurrently at a node (see Figure 3). It is assumed that the TM at a node guarantees the serializability and failure

atomicity of its subtransactions.

Let T_1 be a transaction and let T_{11} and T_{12} be its subtransactions at nodes 1 and 2 respectively. It can clearly be seen that the failure atomicity of T_{11} and T_{12} is not a sufficient condition for the failure atomicity of T. Both T_{11} and T_{12} should succeed or both of them should fail. A "commit" protocol brings about such a decision and it is a standard feature of distributed database systems.

A key feature of the concurrency control suggested in this thesis is that it uses the commit protocols invoked by individual transactions to establish their serializability. This will be further explained in the subsequent paragraphs. The serializability of subtransactions at a single node is termed the local serializability.

Assume that the distributed system has a set of fully synchronized physical clocks [Lam78] to establish the time at which an action occurs (these clocks are used as a proof mechanism and are not required in an actual implementation). CLOCK(a) denotes the physical time at which an action a occurs. Two actions at the same node have different clock values. If a and b are two actions in the node then CLOCK(a) $\neq$ CLOCK(b). Conversely, if for two actions a and b, a $\neq$ b, CLOCK(a) = CLOCK(b) implies a and b occur at different nodes. Physical ordering of two events at a node, written a $<_t$ b implies that CLOCK(a) is less than CLOCK(b). In the definition of the schedule, a < b implies a $<_t$ b.

The one-phase commit protocol is used to terminate distributed transactions. The protocol assumes the existence of a known coordinator and is briefly described:

1. Each participant, when it has completed, records the changes in permanent storage (usually the log file) and sends the "ready to commit" message to the coordinator.
2. When the coordinator has received "ready to commit" from all participants, it sends "commit" messages to all of them. If any participant wanted to abort, the coordinator sends "abort" messages to all participants instead of the "commit" message.
3. Each participant aborts or commits a transaction depending on the message from the coordinator.

Let commit-T_i denote the action that commits the transaction T_i at a node. We assume that it occurs atomically at a node (see Chapter 5). Let CLOCK(commit-T_i) denote the time which this action occurs at a node.

Definition 4:

A schedule S for a single node database is strictly serializable if and only if the following conditions are satisfied:

1. Its dependency graph G is acyclic.
2. If (T_i, T_j) is an edge in G then

$$\forall b \in T_j, \exists a \in T_i: (a < b) \Rightarrow \text{commit-}T_i < b.$$

Strict serializability means that a conflicting request is never granted to a transaction until all the transactions that hold the resource have terminated. Most database

systems support only strict serializability for recovery reasons.

Let $\Pi = \{ T_1, T_2 \ldots T_n \}$ be the set of transactions accessing the DDB. Each T_i, $i = 1,2,\ldots n$ is again a set of subtransactions.

Let $T_i = \{ T_{ix}, T_{iy} \ldots \}$, $x \neq y$, where x,y... denote nodes at which subtransactions are run. T_i is called a distributed transaction if $| T_i | > 1$.

Definition 5:

A global schedule $S = (A_s, <)$ of Π is a set of all actions in each $T_i \in \Pi$ together with a relation $<$ such that conflicting actions are ordered.

Assertion 1:

Two subtransactions of the same transaction never conflict.

Proof:

The sets of resources required by each subtransaction are disjoint because a subtransaction runs only at one node. Hence there can not be a conflict. ■

If a subtransaction sends a message and another receives it, a dependency is introduced between them. However, this dependency is embedded in the semantics of the *send* and *receive* operations. In subsequent discussions, we will assume that such intra-transaction dependencies are automatically upheld in any schedule and that two subtransactions never end up in a deadlock, waiting for messages from each other.

Theorem 2:

Two transactions $T_i, T_j \in \Pi$ conflict only if there exists a k such that T_{ik} and T_{jk} conflict.

Proof:

Since T_i and T_j conflict, there exists an action a in T_i and an action b in T_j such that $a < b$. But we previously asserted that $a < b \Rightarrow$ a and b occur at a single node. Let that node be k. We have only one subtransaction of each transaction at a node. Therefore, $a \in T_{ik}$ and $b \in T_{jk}$. Hence, $a < b \Rightarrow T_{ik}$ *conflict* T_{jk}. ■

Theorem 3:

Let S be a schedule for a set of transactions accessing the DDB in which each node guarantees strict local serializability. Then, commitment of distributed transactions by the one-phase commit protocol is a sufficient condition for serializability of S.

Proof:

Since message delays in the network are variable, the subtransactions of a distributed transaction receive "commit" messages at different clock times. Let CLOCK(min(commit-T_i)) and CLOCK(max(commit-T_i)) denote the earliest and latest clock times at which some subtransaction of T_i committed. Then, $\forall k$, $T_{ik} \in T_i \Rightarrow$ CLOCK(min(commit-T_i)) $\leq$ CLOCK(commit-T_{ik})) $\leq$ CLOCK(max(commit-T_i)).

Assume that the dependency graph of S contains a cycle. Let $T_1, T_2 \ldots T_n, T_1$ be that cycle.

Since (T_1,T_2) is an edge in G, according to theorem 2, $\exists k$ such that T_{1k} and T_{2k} conflict.

Since we have assumed strict local serializability,

CLOCK(commit-T_{1k}) < CLOCK(b), where b is some action in T_{2k}.

i.e. CLOCK(min(commit-T_1)) < CLOCK(b)

i.e. CLOCK(min(commit-T_1)) < CLOCK(min(commit-T_2)).

We can extend the same arguments for each edge and finally obtain

CLOCK(min(commit-T_1)) < CLOCK(min(commit-T_2))..... < CLOCK(min(commit-T_1)), which is impossible.

Hence there can not be a cycle in the graph and S is serializable. ■

This proof of serializability is similar to the proof for two-phase locking [EGL76,TGG82]. The acyclic property of the dependency graph is established based on clock values of commit actions in the proof above. In two-phase locking, this property is established based on the instant at which a transaction holds all its locks.

3.3.1 Multiple Copy Update

The concurrency control algorithm just described works well for a strictly partitioned DDB. However, replication of data needs special treatment. Since only frequently used data will be replicated at different nodes, transactions can easily end up in a deadlock. Several solutions have been proposed to the multiple copy update problem. Some of them are discussed below:

1. Unanimous Update:

   A read request is transformed into a read request on a local copy (if available). A write request is transformed into a set of write requests, one for each copy of the replicated data. Unanimous acceptance of the update by all the copies is necessary before any one of them is updated. If p is the probability that one copy is available, then the probability of success of this update strategy is p ** N, where N is the number of copies. Hence, a write request is less likely to succeed with multiple copies. Another disadvantage of this method is that the deadlock frequency is high.

2. Single Primary Update:

   One copy is designated as primary and the remaining copies are designated as secondaries. Write requests update the primary copy which then propagates the changes to the secondaries. Updates succeed only if the primary copy is available. Another problem with this solution is that the secondaries do not always contain the most recent version of the data, resulting in incorrect reads [Sto79].

3. Single primary-multiple backup:

   This method is a variation of the previous technique and it provides greater resiliency to failures. Write requests update the primary as well as the backups. When the primary fails, the first node in the succession of backups takes over [AlD76].

4. Synchronous Voting:

   Upon arrival of a transaction, the TM, if it has not already initiated voting on a previous transaction, broadcasts a vote for itself to all other TMs. The TMs send messages to each other to establish a partial ordering of nodes, which is then used as the basis for conflict resolution at all nodes [Hol81].

5. Majority Consensus:

   An update succeeds only if it is accepted by a majority of TMs. A version number is then assigned with a particular majority which agreed on a set of updates. Only one version can hold the majority at one time. When a node fails it causes a new majority to be elected. Only a subset of the nodes, known as the ruling majority, with the most recent version of the data, are allowed to vote [Tho79].

Any one of these protocols can be used in the proposed model. The choice depends upon the specific design. Primitives can be defined at the distribution layer depending on the algorithm selected. Multiple copy update algorithms are often complex and difficult to verify [Hol81,Sel81]. Primary copy update is probably a good idea, since it is easy to implement.

3.3.2 Nested Transactions

It has so far been assumed that actions in a transaction, like read and write are atomic. In practice,

very few actions are atomic. It is preferable to have actions performed directly by hardware which would make them atomic (like a single instruction). Even actions are composed of a number of primitive steps and what is an atomic action at one level is a transaction at another level of abstraction. This concept can be extended to the user interface to include user transactions which are nested to an arbitrary depth.

Moss has proposed an implementation of nested transactions in a distributed system [Mos81]. His proposal is the basis for the new distributed computing language Argus, being developed at MIT. His concurrency control algorithm provides synchronization within a transaction and serializes access to objects by two subtransactions of the same transaction. Two-phase locking is used for concurrency control, with some changes to accommodate parent-child relationships. Transactions possess a lock in two ways - *held* and *retained*. A transaction *holds* a lock in a particular mode if it has requested it. It *retains* a lock if the lock is passed to the transaction by a child. The locking rules are:

1. A transaction may *hold* a lock in write mode if no other transaction *holds* the lock (in any mode) and all *retainers* of the lock are superiors of the requesting transaction.
2. A transaction may *hold* a lock in read mode if no other transaction *holds* the lock in write mode and all

retainers of write locks are superiors of the requesting transaction.

3. When a transaction aborts, all its locks (*held* and *retained*, of all modes) are simply discarded. If any of its superiors *hold* or *retain* the same lock, they continue to do so, in the same mode as before the abort [Mos81].
4. When a transaction commits, all its locks (*held* and *retained*, of all modes) are inherited by its parent (if any). This means that the parent *retains* each of the locks (in the same mode as the child *held* or *retained* them).

In the last rule an union of some lock modes is performed. The basic modes are ordered as shown below:

```
none < read < write.
parent's new retained mode =
    max( parent's old retained mode,
         child's retained mode,
         child's held mode ).
```

A transaction's held and retained modes are independent and each separately obeys the rule that the mode never decreases. Transaction management based on these rules is complex and involves substantial message communications.

The proof of serializability established by Theorem 3 can easily be extended for a restricted case of nested transactions. Let $\Pi = \{ T_1, T_2 \ldots T_n \}$ be a set of transactions. Each $T_i \in \Pi$ contains some actions and some

subtransactions, which in turn contain actions and other subtransactions. Each subtransaction runs entirely at one node. We will place the additional restriction that no two descendents of a parent run at a single node (see Chapter 2). The sets of entities acted upon by two related transactions are disjoint. Each $T_i \in \Pi$ is represented by an execution tree reflecting parent-child relationships.

The one-phase commit protocol for a nested transaction is as follows: Each node in the tree decides to enter the commitment phase and sends a "ready to commit" message to its parent only if it has received "ready to commit" messages from all its children. The coordinator (root node), if it has received "ready to commit" message from all its children, commits the entire transaction by sending "commit" messages directly to all its descendents. These messages can also be sent hierarchically down the tree. Since message delays are variable but bounded, all subtransactions will eventually commit. This protocol is vulnerable to failures. More phases in the commitment protocol are necessary to make it resilient to failures.

Theorem 4:

Let S be a schedule for Π. If each node guarantees strict local serializability, the commitment of nested transactions by one-phase commit protocol is a sufficient condition for serializability of S.

Proof:

Identical to that of Theorem 3. ■

Transactions in a hierarchical DDBMS are nested and obey the restrictions mentioned before. This theorem establishes that concurrency control in a hierarchical DDBMS is no different from that in a system without hierarchy.

If the DDB has an hierarchical interconnection, the set of entities known to a node *i* is the union of such sets known to its sons. Subtransactions at the node *i* create new subtransactions at its sons. Since they all operate on disjoint sets of data, Theorem 4 can be applied to establish their serializability.

If two subtransactions are permitted to operate on overlapping sets of objects, some synchronization between them is necessary. TMs will have to know how two subtransactions are related to each other. A large number of new problems are posed, like how conflicts are to be defined, how a subtransaction and its descendent should communicate and how the failure of a node is to be handled. A complete design of such a system is necessarily very complex.

Replication of data is not allowed between two DDBs in the hierarchy (see Chapter 2). Otherwise, imagine that A and B are two DDBs in the hierarchy that have copies of a replicated data item. Two subtransactions of a transaction started at the least common ancestor of these DDBs can conflict at A or B. This introduces a dependency modeled by a directed graph (instead of a tree) and it is forbidden

under the assumptions made before.

3.3.3 Loosely Coupled Systems

Let D_i be a node that is loosely coupled to a distributed database DDB_1, resulting in DDB_2. Let T be a transaction for DDB_2. As defined in Chapter 2, the subtransaction of T that executes at D_i can be collapsed into a single action c, which is commutative. By definition, commutative actions never conflict because there is no dependency between them.

Let $\Pi = \{ T_1, T_2, \ldots T_n \}$ be the set of transactions accessing DDB2. Let $S_2 = (A_2, <_2)$ be a schedule for Π and let A_i be the set of all actions in A_2 which occur at node i. Construct $S_1 = (A_1, <_1)$ where A_1 is the difference of A_2 and A_i. There is no dependency between actions in A_i because they are commutative. Hence $<_1 = <_2$.

Theorem 5:

S_2 is serializable iff S_1 is serializable.

Proof:

The sets of nodes in the dependency graphs of S_1 and S_2 are identical (transactions in Π). Since $<_1 = <_2$, no new dependency is introduced between transactions in Π because of actions in A_i. Consequently, the sets of edges in in the dependency graphs of S_1 and S_2 are also identical. Hence these graphs are identical. ■

This theorem establishes concurrency control in a loosely coupled system. Its implication is that the loose

coupling of D_i to DDB_1 does not affect concurrency control in DDB_1.

3.4 Summary

In this Chapter, two fundamental existing algorithms - two-phase locking and timestamp ordering - are described. It is proved that local serializability, together with commit protocols, is a sufficient condition for global serializability in the proposed model. This proof is extended for nested transactions and loosely coupled systems.

Chapter 4

Deadlock

4.1 Centralized Systems

Deadlock is a fundamental problem to be addressed in any system of concurrent processes which require exclusive access to shared resources. A well known example is the deadlock which occurs in operating systems, when chunks of memory are allocated to processes. Consider two processes P_1 and P_2, which have 20K bytes of memory allocated to them. Suppose that the total memory available on the system is 64K bytes and that both P_1 and P_2 request an additional 30K bytes of memory each. Neither P_1 nor P_2 can be allocated the requested memory, and in the absence of any arbitration, these processes will wait for ever. Operating systems usually have facilities to recognize and resolve such conditions. In the example above, one of the processes could perhaps be swapped to disk, freeing its memory.

Similar situations arise in database systems, when transactions require exclusive access to stored entities. Consider two transactions T_1 and T_2 active in the system. If T_1 requests a conflicting lock on an entity which is held by T_2, it is forced to wait until T_2 releases the lock. T_1 is then said to *wait-for* T_2. T_2 itself may be waiting for another transaction, and a deadlock exists if this *wait-for* relationship extends to a cycle. Deadlock situations can be characterized by *wait-for* graphs [Hol72]. A wait-for graph

is an immediate consequence of the dependency graph of a set of transactions (see Chapter 3).

Let $D = \{E_1, E_2, \ldots E_k\}$ be the database and let $\Pi = \{T_1, T_2, \ldots T_n\}$ be the set of transactions accessing it. Two actions $a_1 \in T_1$ and $a_2 \in T_2$ are said to conflict if they both operate on the same data and one of them is a write. This condition is usually detected by locking, where a lock has to be obtained on an entity before an action can use it. A transaction can request several such locks and hence it can simultaneously wait for several transactions to complete.

The wait-for graph (also called the system graph) G(W,U) is defined such that:

1. $W = \{ T_i \mid T_i \in \Pi \}$
2. $U = \{ (T_i, T_j) \mid T_i, T_j \in \Pi$ and $(T_i$ *waits-for* $T_j) \}$

A deadlock exists iff G does not contain a cycle [Hol72].

4.1.1 Basic Approaches to Deadlock Handling

The three basic approaches to handle deadlocks are prevention, avoidance and detection.

Prevention:

Prevention techniques restrict the usage of resources in such a way that deadlocks can not occur. One strategy is to disallow dynamic resource requests - all the resources needed are requested before a transaction is initiated. The system checks to see if all the resources requested are available. If not, the transaction is forced to wait until

such a time. This means that the read and write sets of a transaction should be known in advance, a situation often not possible in database systems.

Another prevention strategy is preempting a blocked transaction when an active transaction requests a resource held by it. The disadvantages of this approach are the preemption of transactions even when there is no deadlock and cyclic restart. In cyclic restart, a transaction goes through several phases of preemption and restart, without ever being completed.

Another prevention technique is to order the resources and force all transactions to request the resources in the ascending (or descending) order. Essentially, each transaction has a priority, which is the rank of the highest order resource it has requested. A transaction is allowed to wait for only higher priority transactions.

Avoidance:

In avoidance techniques, advance information about resource usage is used to control allocation such that there is at least one way in which all transactions can complete. For example, an avoidance technique suggested by Lomet [Lom79] builds what is termed the *potential* wait-for graph using the *resources claimed* and *resources held* sets of all transactions. The *resources claimed* set of a transaction contains resources whose use is anticipated in future. An allocation is considered safe only if it does not create a cycle in the potential wait-for graph.

Detection:

Detection techniques allow transactions to make resource requests freely. A resource, if it is available, is allocated to the first transaction that requests it. Deadlocks are detected by building the wait-for graph. If there is a cycle in the graph, a victim is chosen and aborted, thus breaking the cycle.

Associated with each transaction is a cost which measures the overheads involved in aborting it. When a deadlock is found, the victim chosen is the transaction with the least cost. When there are multiple cycles, a set of transactions which will break the cycles are chosen as victims. Victims are always aborted and all resources held by them are released. Isloor and Marsland have provided a summary of these techniques [IsM80].

4.1.2 Probability of Deadlock Occurrence

Let there be N transactions in the system, each requiring r resources from an universe of R resources ($r << R$). Gray et al. [GHO80] used an empirical model to show that the probability of deadlock is equal to $(N * r^4) / (4 * R^2)$. Their conclusions, based on this analysis and actual observations, may be summarized by:

1. Probability that a transaction experiences deadlock is proportional to the degree of concurrency (N).
2. Waits rise as the second power of transaction size.
3. Deadlocks rise as the fourth power of transaction size.

4. Almost all deadlock cycles are of size 2 [GHO80].

4.2 Deadlock in Distributed Systems

The deadlock detection and prevention techniques for centralized systems can be extended for distributed systems [MaI80,KKN83]. However, these solutions are very complex because of the distribution of control.

4.2.1 Detection Techniques

In distributed systems, transactions running at several nodes can find themselves deadlocked. A deadlock may exist in the system even if no single node can detect it. For example, consider two transactions T_1 and T_2, at nodes N_1 and N_2 respectively. T_1 holds the exclusive lock on entity e_1 at N_1, and requests the exclusive lock on entity e_2 at N_2. T_2 holds the exclusive lock on e_2 and requests the lock on e_1. T_1 and T_2 are deadlocked, and controllers at N_1 and N_2 will not be able to recognize it independently.

Each node constructs a wait-for graph (called the local graph) using transaction waiting conditions known to it. The global wait-for graph (called the global graph) is the union of all local graphs. The distributed deadlock detection problem is reduced to the problem of finding cycles in the global graph. The existing distributed deadlock detection algorithms can be classified according to how they construct this graph [KKN83]. The global graph may be constructed at one node, at all nodes, or partially constructed at each

node.

In the centralized methods [MeM79,Gra79,HoR82], one node is elected to detect distributed deadlocks. Each node resolves local deadlocks internally and reports wait-for conditions of distributed transactions to the elected node, which constructs the global graph by merging local graphs.

Another solution [MaI80] involves construction of the global graph at all nodes. Local deadlocks are resolved internally at each node and the global deadlocks are recognized at all the nodes (since the wait-for conditions of distributed transactions are known to all nodes). An event which can change the status of a distributed transaction (like an allocation or a release of a resource) is communicated immediately to all other nodes. The global graph is maintained by incremental changes.

The message-passing deadlock detection algorithms belong to the category in which the global graph is partially constructed at each node. In the original scheme proposed by Goldman [Gol77], a process is allowed to wait for only one other resource and hence only one other process. Suppose that process p_1 is waiting for process p_2. p_1 sends a message (also called a probe in this context) to p_2. If p_2 is active, it destroys the message from p_1. Otherwise, if p_2 itself is waiting for another process, say p_3, p_2 appends its name to the message from p_1 and sends it to p_3. If the message eventually reaches p_1 again, there is a deadlock in the system. Actual details of implementation

and the format of the messages have been omitted here. The major disadvantages of this scheme are:

1. A process can wait for only one resource at a time.
2. Several controllers can detect the same deadlock, thereby complicating resolution.

Two solutions have been proposed [ChM82,JaV82] to remove the first restriction. A detection technique based on the partial construction of the global graph was proposed by Menasce and Muntz [MeM79]. However, it was shown to be incorrect by Gligor and Shattuck [GlS80]. The improvement suggested by Gligor and Shattuck was subsequently shown to be incorrect by Tsai and Belford [TsB82].

The nature of the communication network introduces two problems -- detection of false deadlocks and delayed detection of existing deadlocks. False deadlocks are detected when the conditions that caused the deadlock have changed during the construction of the global graph. For example, a transaction might have been aborted, releasing all its resources; unknown to the detection algorithm, a node may have crashed, aborting all transactions that run there.

Periodic transmission of the local graphs is the reason for delayed deadlock detection. Incremental construction of the graph [MaI80] reduces delays in deadlock detection.

Some detection techniques consider the global graph itself as a distributed database, cycle detection as a query

and changes to the global graph as update transactions. Synchronization methods like timestamp ordering are used update the global graph. Based on this approach, several variations are possible in the deadlock detection algorithms and a few have been proposed already [TsB82,KKN82].

4.2.2 Prevention Techniques

Time-outs and timestamps are the frequently used prevention techniques in distributed systems. Time-outs are used to revoke transaction-waits after a predetermined time interval. When a transaction makes a request, the system sets a timer for it with a specific value. The transaction is aborted if the request is not granted before the timer runs out. Though it is simple to implement, the basic problem in this scheme is the choice of the time-out interval. If it is too high, a deadlock may persist for a long time. If it is too small, too many transactions will get aborted. When the system is congested, too many transactions will be aborted and restarted, further adding to congestion.

Two methods, called *wound-wait* and *wait-die* have been proposed [RSL78] for distributed deadlock prevention based on timestamps. In both methods, the priority of a transaction is determined by its timestamp. *Wait-die* is a non-preemptive technique. Suppose that a transaction T_1 tries to wait for T_2. If T_1 is younger than T_2 it is permitted to wait. Otherwise, T_1 is aborted and is forced to

restart. It is important that T_1 is not assigned a new timestamp when it restarts. In the *wound-wait* technique, T_1 is allowed to wait if it is older than T_2. Otherwise, T_2 is aborted. Both *wound-wait* and *wait-die* avoid cyclic restarts. An undesirable property of *wait-die* system is that a transaction which dies may restart again and end up in the same conflict which forced it to abort previously. However, it has a desirable property that once a transaction has obtained all the locks it needs, it will not be forced to restart. It has been suggested [RSL78] that *wound-wait*, combined with the knowledge of commitment of transactions, forces fewer restarts in total.

4.3 Deadlock Handling in the Proposed Model

Any of these deadlock prevention/detection strategies can be used in the transaction management model proposed in Chapter 2. Some details unique to this model are outlined below:

In the proposed model, a transaction T_1 never requests access to a remote entity. Rather, it creates a subtransaction, say T_{11}, at the remote node and communicates the request to T_{11}. T_{11} reads the entity and performs the changes necessary. At each node we have a set of independent transactions, some of which are active while the others are waiting. Such a wait is called *resource-wait*. Parent-child relationships between transactions introduces a different type of waiting condition. This wait is called the

session-wait [Gra79]. In the example above, T_1 and T_{11} are involved in a session-wait. Local deadlocks entirely contained within a node are handled by the TM of the local database machine (see Chapter 2). Local deadlocks are of the form:

resource-wait->....resource-wait->....resource-wait.

Global deadlocks involving multiple nodes are of the form:

...resource-wait->....session-wait->resource-wait...

..->resource-wait [Gra79].

Note that a basic assumption of the proposed model is that there are no deadlocks within a transaction.

The distribution layer at a node, say N_1, is unaware of the waits-for relationships between subtransactions at that node. To obtain the waits-for information from the local TM, the distribution layer invokes the following primitive:

wait-status(Π) *returns*(G_1)

Π is the set of all subtransactions (of distributed transactions) running at N_1. G_1 is called the Reduced Local Graph (RLG) of N_1, and is defined as follows:

$G_1(W,U)$:

$W = \{ t \mid t \in \Pi \}$

$U = \{ (t_1,t_2) \mid t_1,t_2 \in \Pi$ and t_2 is reachable from t_1 in the local graph at $N_1 \}$

Note that G_1 is acyclic, since all local deadlocks are resolved internally at a node. This may cause a distributed transaction to be aborted in preference to a purely local transaction. This is a disadvantage of the proposed model.

Global deadlock is detected by merging RLGs from all nodes. For example, in centralized methods, a node is periodically elected to receive RLGs. The elected node glues together the RLGs, looks for cycles, selects victims and informs the nodes concerned.

A node may place restrictions on (sub)transaction waits. For example, it may assign priorities and forbid higher priority transactions to wait for lower priority transactions. Such details are purely internal to the node. However, if a subtransaction is allowed to wait for another, it should be reported when the *wait-status* primitive is invoked.

Gray et al. [GHO80] have hypothesized that the global graph is usually very sparse and that the probability of deadlocks is very low. Hence, they have advocated the use of message-passing deadlock detection algorithms.

4.3.1 Hierarchical DDBMS

As pointed out before, each node in the proposed model can by itself be a DDBMS. This permits the system to be modeled by a tree whose nodes are the DDBMSs. Each DDBMS resolves deadlocks within it internally and constructs the reduced local graph for subtransactions initiated by its parent. This graph is sent to the parent when requested (the *wait-status* primitive).

The execution tree of a transaction corresponds to a subset of the system structure. Suppose that there are two

transactions T_1 and T_2 such that the descendent of T_1 waits for the descendent of T_2 at a DDBMS, say d_1, and that the descendent of T_2 waits for the descendent of T_1 at some other DDBMS, say d_2. T_1 and T_2 are deadlocked and this deadlock is detected by the least common ancestor of d_1 and d_2 [HoR82,MeM78].

Note again that two descendents of a transaction never reside at a single node. This prevents conflicts within a transaction. However, if general purpose nested transactions are allowed without this restriction, a deadlock can develop within a transaction. One proposed solution [Mos81] to this problem is an extension of Goldman's algorithm.

4.3.2 Loosely Coupled Systems

The deadlock-prevention algorithm suggested by Ries [Rie82] is well suited for loosely coupled systems. This algorithm distinguishes between atomic and non-atomic sessions. When a transaction initiates a subtransaction, it is said to be in an atomic session if:

1. No communication is necessary between them (no send or receive primitives within the subtransaction)
2. Commitment of the subtransaction is independent of its parent.

Otherwise, the session is defined to be non-atomic.

Let D_i be the database loosely coupled to the distributed database DDB_1. By our definition of loosely

coupled systems, the initiation of a subtransaction for D_i results in an atomic session.

Three protocols were suggested by Ries for deadlock prevention. These protocols impose restrictions on the initiation of sessions. It has been proved [Rie82] that if all TMs obey these protocols, there will not be a global deadlock in the system.

Note that by definition, the loose coupling of DDB_1 and D_i is asymmetric. That is, DDB_1 acts as a producer and consumer of transactions whereas D_i acts only as a consumer.

Assertion:

Let D_i be loosely coupled to DDB_1. Then, all deadlocks in the system are entirely contained within DDB_1 or D_i.

Proof:

A subtransaction at D_i never waits for a transaction at DDB_1 because:

1. D_i acts only as a consumer.
2. A subtransaction at D_i is involved only in atomic sessions with its parent.

A cycle between two transactions in DDB_1 and D_i is not possible because there are no edges pointing from D_i to DDB_1. Hence, cycles, if any, must be entirely within DDB_1 or D_i. ■

The implication of this assertion is that no strategy is required to handle deadlocks between DDB_1 and D_i.

4.4 Summary

Deadlock detection and prevention techniques in centralized and distributed systems are described in this Chapter. These techniques are extended for the proposed transaction management model.

Chapter 5

Recovery

The aspects of recovery which affect transaction management in the proposed model are described in this Chapter. Existing techniques can be used in the model without any significant changes. Hence, this Chapter contains only a summary of the published recovery techniques.

Reliability is often quoted as one of the major advantages of a distributed system. However, it is only a potential advantage. There are several problems to be overcome before the desired reliability can be achieved. A variety of failures are possible in a distributed system and it deviates from its expected behavior when a failure occurs. Depending on the level of reliability desired, the types of failures to be handled must be decided [Gar80]. The problem of recovery can be classified into 3 subproblems:

1. Termination of active transactions when a failure is detected.
2. Continuation of (possibly degraded) processing during the failure
3. Restart and resumption of normal processing after the source of failure has been removed.

The recovery protocols of centralized systems will be described first.

5.1 Centralized Systems

The types of failures expected in a centralized system are:

1. Media failure:

   A media failure occurs when an unrecoverable error is detected in the file system.

2. Transaction failure:

   A transaction failure occurs when it has to be aborted because of incorrect computations or invalid access requests.

3. System failure:

   A system failure occurs when a serious error occurs in the system. An example is the failure of the processor.

Media failures are usually handled by lower level protocols. A layered architecture can be used to construct reliable file systems [Ver77,Ver78,Ver79,Lam81]. Protocols supported by these models handle media failures internally, without the cooperation of the transaction management layer.

Transaction and system failures affect the atomicity of transactions. The following recovery protocols are commonly used to handle such failures [Gra79].

Consistency Lock Protocol:

A transaction can be considered as a set of actions on *recoverable* objects. An object is recoverable if its state can be restored to some previous value. For example, database entities are recoverable, while the state of a cash dispensing teller is unrecoverable. A log (also called an

audit-trail or a journal) is a file that records the history of state changes to the database entities. Updates to the database are recorded in the log, along with the identity of the transaction that caused the change.

A transaction does not directly update entities in the permanent memory. It changes only the copies stored in its private workspace in volatile memory. When a transaction completes its computations, it requests what is known as *commitment*. Commitment is a process that updates entities in the permanent memory. From the transaction's point of view, commitment is an unrecoverable, atomic action. Note that the changes made by a transaction are made known to others only after its commitment.

If a transaction is aborted before requesting commitment, its actions are undone simply by discarding its private workspace and releasing all the locks held by it. However, once it requests commitment, the system must guarantee the failure atomicity of commitment. The Write Ahead Log (WAL) protocol is used for this purpose.

WAL Protocol:

Before committing a transaction, the system posts a history of its updates to the log file and forces the log to permanent memory (that is, it is not kept in the volatile memory, waiting to be transferred to disk) [Gra79]. WAL implements the *secure-transaction* primitive described in Chapter 2.

Undo-Redo Protocol:

The commitment of a transaction proceeds in two successive phases -- first the WAL protocol, followed by actual changes in the database. If the system crashes during phase 1, the transaction is aborted and any entries corresponding to it in the log are ignored. If the system crashes during phase 2, partially completed updates are undone and phase 2 of the commitment is carried out from the beginning [Gra79].

Checkpoints:

A checkpoint is a system state when no transaction is active. Hence, it represents a consistent snapshot of the system state. The system forces checkpoints of its state periodically. Techniques are available to record checkpoints while transactions are still active [Ver78,Kus82], by relaxing the condition mentioned above. Checkpoints are usually recorded in the log file with sufficient information to restore the system state [McD81]. Restart protocols scan the log from the latest checkpoint to identify partially completed transactions.

5.2 Issues in Distributed Systems

We deal with the distributed system at a high level of abstraction and assume that the following types of failures do not occur:

1. No malevolent failures:

   A node does not act maliciously; it always follows the system protocols.

2. No undetected failures:

   The existence of a very reliable virtual network, like the RelNet of SDD-1 [HaS80,BRS80], is assumed. Such a network detects all system failures and reports them to higher levels (the RelNet goes a step further and supports facilities like the global clock and guaranteed message delivery). A design based on such a network has been strongly advocated [HaS80] for systems with high reliability requirements.

3. No communication failures:

   The network is very reliable and is assumed to satisfy the message transmission properties outlined in Chapter 2 [Gar80].

The following types of failures have been addressed in this report:

1. Transaction failures.
2. Single and multiple node failures.
3. Network Partitions.

A partition occurs in the network when the nodes are divided into two or more disjoint groups such that a node can communicate only within its group. Network partitioning is one of the severest disasters that can befall a distributed system [AlD76].

5.2.1 Termination Protocols

A distributed transaction executes at several nodes and may change the states of databases at these nodes. Several

failures are possible during its processing. For example, a subtransaction may fail or a node may crash. To preserve the consistency of the database, it is essential that the transaction be terminated uniformly at all the nodes. That is, all subtransactions of a transaction commit or all of them abort. Two-phase commit is the standard protocol that is used to bring about such a decision.

Two-phase Commit Protocol:

The node at which a transaction originated is usually selected as the coordinator for the protocol. Each subtransaction, when it has completed its work, sends a "completed" message to the coordinator. The protocol proceeds according to the following steps:

1. When the coordinator has received "completed" messages from all subtransactions, it sends "prepare to commit" messages to all of them. If any subtransaction had failed, the coordinator recognizes this condition and sends "abort" messages to all of them.
2. Each subtransaction receives the "prepare to commit" message and records its updates in the log file (*secure-transaction* primitive and the WAL protocol) and sends a "prepared" message to the coordinator. If for any reason, a subtransaction chooses to abort, it sends a "abort" message instead.
3. The coordinator, if it has received "prepared" messages from all subtransactions, sends "commit" messages to all of them. Otherwise, it sends "abort" messages.

4. Each subtransaction commits its results after it receives the "commit" message from the coordinator [Gra79,Lam81].

The protocol is called "two-phase" because of the two distinct states of a subtransaction during its commitment. The first phase extends from the time a subtransaction completes till the time it receives the "prepare to commit" message. The second phase exists between the instants at which it receives the "prepare to commit" and "commit" messages. It is easy to see that the protocol consistently terminates the subtransactions in the absence of any failures. Suppose that the coordinator or a subtransaction failed during the first phase. Other nodes can detect this condition and switch to "abort" independently.

If a subtransaction had failed during the second phase, it can not reach a decision about commitment when it restarts. It either has to talk to the coordinator or one of the subtransactions to find out the decision taken.

If the coordinator fails during phase two, we have additional problems. For example, it may have sent "commit" messages to some but not all subtransactions. A subtransaction which has not received the "commit" message has the following options:

1. Talk to other subtransactions and find out if any one of them has received the "commit" message. If so, commit the subtransaction. Otherwise wait till the coordinator restarts.

2. Wait till the coordinator recovers.

The first option involves additional communications and forbids *independent recovery*. Independent recovery of a subtransaction is the process by which it can reach a decision solely based on its current state, without talking to other subtransactions [SkS81]. The second option is said to be a "blocking" option; it forces an operational node to wait for a failed node [SkS81].

The second phase of the two-phase commit protocol is called the "in-doubt area" [Bal81] or the "window of uncertainty" [Coo82]. It is possible to introduce additional phases in the commit protocol to reduce the window of uncertainty. Skeen [Ske81,SkS81] used a formal model of transaction commitment to show that any number of phases in the protocol will not make it non-blocking to multiple (two or more) node failures or network partitions.

A large number of commitment protocols have been proposed in literature. Some of them are variations of the two-phase commit protocol while the others try to handle a specific failure with procedures that involve excessive overheads during normal processing.

Distributed Checkpointing:

Distributed checkpointing is used to roll back the entire system to an earlier consistent state. A system wide roll-back may be necessary to recover from a catastrophe. A related problem is the identification of recovery lines in a system of concurrent processes. Each of these processes sets

its checkpoints independently. A recovery line is a set of checkpoints, one at each process, such that no single transaction initiated by a process is active before and after these checkpoints. A number of algorithms already exist for distributed checkpointing and detection of recovery lines [Fer81,FGL82,Kus82,McD81,RLT78,TKN81].

5.2.2 Processing during failures

Since the components of a DDBMS have independent failure modes, failure of a node should not seriously affect the functioning of operational nodes. For example, a transaction should never be delayed because of failures unless the only copy of data requested by it resides in a failed node. In case of partitioning, at least one partition should be allowed to proceed. Some solutions have been suggested [DaG81,Bha82] for transaction processing during failures.

The research on distributed recovery has been concentrated on termination protocols. Existing solutions to transaction processing during failures are application dependent and are not general. We suggest that a formal model be developed for distributed transaction processing during failures, one which includes single and multiple node failures and network partitioning.

5.2.3 Hierarchical and loosely coupled systems

The two-phase commit protocol can easily be extended for a tree of transactions in a hierarchical DDBMS. If the transaction at the root of a tree knows about all its descendents, no changes are necessary to the protocol. If the messages always flow down the tree, a small change is necessary. A node sends the "completed" and "prepared" messages of the protocol to its parent only if it has received such messages from all its sons; it sends the "prepared to commit" and "commit" messages of the protocol to all its sons when it receives these messages from its parent.

So far, there has not been a necessity for commitment and recovery protocols for a tree of transactions. Hence, no work seems to have been reported about the potential problems and the performance of these protocols. The problems of recovery in loosely coupled systems include the generation and processing of compensating transactions. For example, a compensating transaction itself may fail, forcing retries. This area needs further study.

5.3 Summary

Failure and recovery aspects of transaction management are outlined in this Chapter. The protocols for ensuring failure atomicity of transactions in centralized systems are described first. Failure atomicity of distributed transactions is achieved by termination protocols and

descriptions of some of them are also given.

Chapter 6

Conclusions

A model for distributed transaction management has been proposed in this thesis. The concurrency control problem for distributed database systems was partitioned into 3 subproblems - serializability, multiple copy update and deadlock detection. A major effort was devoted to establishing the synchronization atomicity of a transaction, based on this property of its subtransactions. Existing solutions for multiple copy update and deadlock detection were adapted to make the solution complete.

The essential difference between the proposed model and existing ones is the synchronization technique. Existing models establish the synchronization atomicity of a transaction based on the atomicity of the actions. Whereas, the proposed model establishes the same, based on the atomicity of subtransactions. This feature was shown to be the basis for the extensions to the transaction concept to support nested and commutable transactions.

Since a transaction is divided into subtransactions, a natural extension will be the division of subtransactions into further subtransactions and so on, resulting in a nested transaction. A nested transaction can be modeled by its execution tree. The synchronization atomicity of a node in the tree is derived based on this property of its sons. This has also been called the multi-level atomicity problem in the literature [BeR81].

The proposed model was extended for a restricted case of nested transactions. This extension was shown to support a hierarchical DDBMS. Current systems assume full connectivity between component databases in a DDB. The hierarchical DDBMS groups the components into clusters and the clusters are connected in the form of a tree.

The other extension of the model was termed "loosely coupled systems". The notion of commutable subtransactions was used as the basis for defining the loose coupling of a node to a DDB.

The transaction concept is being extended for non-database, general purpose distributed computing [Lis82,SpS83]. The nodes of the distributed system are assumed to support some objects and procedures that manipulate these objects (abstract data types). We hypothesize that the proposed model, with its extension to nested and commutable transactions, can be used to advantage in such a system.

6.1 Suggestions for future work

1. The tree model for DDBMSs can be extended to a graph. The DDBMS can be thought of as a network of transaction processors and the structure of the graph is determined by the entities known to each transaction processor. For example, if the structure is a directed acyclic graph, the set of entities known to a node may be the union of all such sets reachable from this node. The notion of

nested transactions is closely related to this model and as shown in Chapter 3, synchronization within a transaction may be necessary.

2. Two implementation techniques have been proposed for supporting nested transactions. The first is based on two-phase locking [Mos81] and the second is based on timestamps and multi-version objects [Ree83]. These proposals can be extended. For example, Moss' proposal [Mos81] severely restricts communications between subtransactions and this restriction can be removed (by checkpointing a transaction; one has to read his proposal before understanding the connection between checkpoints and communications).
3. So far, we have assumed that a transaction typically runs for a few seconds or minutes. However, transactions which run for days instead of minutes can be expected [Gra81]. Such transactions are very vulnerable to failures and the costs of aborting/restarting them are very high. A single transaction can be made more reliable by introducing checkpoints within it. The implementation of such checkpoints and the associated problem of finding recovery lines can be handled by the distribution layer. This problem needs further study.
4. No general technique exists for the generation of compensating transactions. While the semantics of recovery in loosely coupled systems has been pointed out, the procedures were not described. This area needs

further study.

5. Only some straightforward extensions of existing deadlock detection algorithms and commit protocols for a hierarchical DDBMS have been described in this thesis. More sophisticated techniques may be necessary.

References

[AlD76] Alsberg, P. A. and Day, J. D., A Principle of Resilient Sharing of Distributed Resources, *Proc. Int. Conf. on Software Eng.* , San Francisco, Calif., Oct 1976, 562-570.

[ALS78] Anderson, T., Lee, P. A. and Shrivastava, S. K., A Model of Recoverability in Multi-level Systems, *IEEE Trans. on Software Eng. SE-4*, 6 (Nov. 1978), 486-494.

[Bad79] Badal, D. Z., Correctness of Concurrency Control and implications in Distributed Databases, *Proc. IEEE COMPSAC*, Chicago, 1979.

[Bal81] Balter, R., Selection of Commitment and Recovery Mechanism for a Distributed Transaction System, *IEEE Symp. on Reliability in Distributed Software and Database Systems*, Pittsburgh, Penn., June 1981.

[BBD82] Balter, R., Berard, P. and Decitree, P., Why Concurrency Control in Distributed Systems is more fundamental than Deadlock Management, *ACM SIGOPS Symp. on Principles of Distributed Computing*, Ottawa, Canada, Aug 1982.

[BeS78] Bernstein, P. A. and Shipman, D. W., A formal model of concurrency control mechanisms for Database Systems, *Proc. 3rd Berkeley Workshop on Distributed Data Management and Computer Networks*, Aug. 1978 .

[BRS80] Bernstein, P. A., Rothnie, J. and Shipman, D. W., The Concurrency Control mechanism of SDD-1: A system for Distributed Databases, *ACM Trans. Database Systems 5*, 1 (March 1980), 18-25.

[BeG81] Bernstein, P. A. and Goodman, N., Concurrency Control in Distributed Database Systems, *Computing Surveys 12*, 2 (June 1981), 185-221.

[BeG82] Bernstein, P. A. and Goodman, N., A sophisticate's introduction to Distributed Database Concurrency Control, *Proc. 8th Int. conf. on Very Large Databases* , Mexico City, Sept. 1982.

[BeR81] Best, E. and Randell, B., Formal model of atomicity in Asynchronous Systems, *Acta Inf. 16*, (1981), 93-124.

[Bha82] Bhargava, B., Optimistic Concurrency Control Approach to Distributed Databases, *Proc. IEEE Symp. on Reliability in Distributed Software and Database*

Systems, Pittsburgh, PA, July 1982.

[Bor81] Borr, A., Transaction Monitoring in Encompass: Reliable Distributed Transaction Processing System, *Proc. 7th Int. Conf. on Very Large Data Bases*, Cannes, France, Sept 1981.

[BrL82] Breitwieser, H. and Leszak, M., A Distributed Transaction Processing protocol based on Majority Consensus, *ACM SIGOPS Symp. on Principles of Distributed Computing*, Ottawa, Canada, Aug. 1982.

[BMR82] Brownbridge, D. R., Marshall, L. F. and Randell, B., The Newcastle Connection or UNIXes of the World Unite, *SoftwarePractice & Experience 12*, (1982), 1147-1162.

[CFL82] Chan, A., Fox, S., Lin, W. K., Nori, A. and Ries, D., The implementation of an integrated Concurrency Control and Recovery Scheme, *ACM SIGMOD Int. Conf. on Management of Data*, Orlando, Florida, June 1982 .

[ChM82] Chandy, K. M. and Misra, J., An algorithm for Detecting Resource Deadlocks in Distributed Systems, *ACM SIGOPS Symp. on Principles of Distributed Computing*, Ottawa, Canada, Aug 1982.

[CoD73] Coffman, E. G. and Denning, P. J., in *Operating Systems Theory*, Prentice Hall Inc., 1973.

[CoB79] Colliat, G. and Bachman, C., Commitment in a Distributed Database, in *Database Architecture*, Bracchi, G. and Nijjsen, G. M. (ed.), North-Holland, 1979.

[Coo82] Cooper, E. C., Analysis of Distributed Commit Protocols, *ACM SIGMOD Int. Conf. on Management of Data* , Orlando, Florida, June 1982.

[DaG81] Davidson, S. B. and Garcia-Molina, H., Protocols for Partitioned Distributed Database Systems, *IEEE Symp. on Reliability in Distributed Software and Database Systems*, Pittsburgh, Penn., June 1981.

[Dav73] Davies, C. T., Recovery Semantics for a DB/DC System, *Proc. ACM National Conference*, Atlanta, 1973.

[Dee81] Deen, S., Architecture of Distributed Database Systems, in *Distributed Data Sharing Systems*, Riet, R. P. V. D. and Litwin, W. (ed.), North Holland, 1981.

[DER82] Devor, C., Elmasni, R. and Rahimi, S., The design of DDTS: A testbed for reliable Distributed Database Management, *Proc. 2nd Int. conf. on Reliability in Distributed Software and Database Systems*, Pittsburgh, PA, July 1982.

[DiS80] Dijkstra, E. W. D. and Scholten, C. S., Termination Detection for Diffusing Computations, *Inf. Proc. Letters 11*, 1 (Aug 1980), 1-4.

[EGL76] Eswaran, K. P., Gray, J. N., Lorie, R. A. and Traiger, I. L., The notions of Consistency and Predicate Locks in a Database System, *Comm. ACM 19*, 11 (Nov. 1976), 624-633.

[Fer81] Ferran, G., Distributed Checkpointing in a Distributed Data Management System, *IEEE Symp. on Realtime Systems*, 1981.

[FGL82] Fischer, M. J., Griffeth, N. D. and Lynch, N. A., Global States of a Distributed System, *IEEE Trans. on Software Eng. SE-8*, 3 (July 1982).

[Gar80] Garcia-Molina, H., Reliability Issues for Completely Replicated Distributed Databases, *Proc. IEEE Fall COMPCON*, 1980.

[GaW82] Garcia-Molina, H. and Wiederhold, G., Read-only Transactions in a Distributed Database System., *ACM Trans. Database Systems 7*, 2 (June 1982), 209-234.

[GlS80] Gligor, V. D. and Shattuck, S., On Detection of Deadlocks in Distributed Systems, *IEEE Trans. on Software Eng. SE-6*, 4 (Sept 1980), 435-440.

[Gol77] Goldman, B., Deadlock Detection in Computer Networks, Tech. Rep.-MIT/LCS/Tech. Rep.-185, Lab of Computer Science, Sept 1977.

[Gra78] Gray, J. N., Notes on Database Operating Systems, in *Operating Systems: An Advanced Course*, Springer Verlag, 1978, 393-481.

[GHO80] Gray, J., Homan, P., Obermarck, R. and Korth, H., A Straw Man Analysis of Probability of Waiting and Deadlock, *IBM Research Report RJ3066(38112)*, San Jose,CA, 1980.

[Gra80] Gray, J. N., A Transaction Model, in *Automata, Languages and Programming* , Springer Verlag, 1980, 282-298 .

[Gra81] Gray, J. N., The Transaction Concept: Virtues and Limitations, *Proc. 7th Int. Conf. on Very Large*

Data Bases, Cannes, France, Sept 1981.

[GrM81] Gray, J. and McJones, P., The Recovery Manager of the System R Database Manager, *Computing Surveys 13*, 2 (June 1981), 223-242.

[HaS80] Hammer, M. and Shipman, D., Reliability Mechanisms for SDD-1: A System for Distributed Databases, *ACM Trans. Database Systems 5*, 4 (Dec 1980), 431-466.

[HoR82] Ho, G. S. and Ramamoorthy, C. V., Protocols for Deadlock Detection in Distributed Database Systems, *IEEE Trans. on Software Eng. SE-8*, 6 (Nov 1982).

[Hol81] Holler, E., Multiple Copy Update , in *Distributed Systems: Architecture and Implementation*, Springer Verlag, 1981.

[Hol72] Holt, R. C., Some Deadlock Properties of Computer Systems, *Computing Surveys 4*, (Sept. 1972), 179-196.

[IsM80] Isloor, S. S. and Marsland, T. A., The Deadlock Problem: An Overview, *Computer*, Sept 1980, 58-77.

[JaV82] Jagannathan, J. R. and Vasudevan, R., A Distributed Deadlock Detection and Resoultion Scheme: Performance Study , *3rd Int. Conf. on Distributed Computer Systems*, Ft. Lauderdale, Fla., Oct. 1982.

[KKN83] Korth, H., Krishnamurthy, R., Nigam, A. and Robinson, J. T., A framework for understanding Distributed Deadlock (Detection) algorithms, *ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, Atlanta , 1983.

[Kus82] Kuss, H., On Totally Ordering Checkpoints in Distributed Databases, *Proc. ACM SIGMOD Int. Conf. on Management of Data*, Orlando, Florida, June 1982.

[Lam78] Lamport, L., Time, Clocks and Ordering of Events in a Distributed System , *Comm. ACM 21*, 17 (July 1978), 558-565.

[Lam81] Lampson, B. W., Atomic Transactions, in *Distributed Systems: Architecture and Implementation*, Springer Verlag, 1981, 246-265.

[LeL79] LeLann, G., An Analysis of Different Approaches to Distributed Computing, *Proc. 1st Int. Conf. on Distributed Comupter Systems*, Huntsville, Alabama, 1979.

[LeL80] LeLann, G., Consistency, Synchronization and

Concurrency Control, in *Distributed Databases*, Draffan, I. W. and Poole, F. (ed.), Cambridge University Press, Sheffield, England, 1980.

[LeL81] Leung, J. Y. and Lei, E. K., On minimum cost Recovery from System Deadlock, *IEEE Trans. on Computers C-28*, 9 (Sept 1981), 671-677.

[Lin80] Lindsay, B. G., Single and Multi Site Recovery Facilities, in *Distributed Databases*, Draffan, I. W. and Poole, F. (ed.), Cambridge University Press, 1980.

[Lis79] Liskov, B., Primitives for Distributed Computing, *Proc. 9th Symp. on Operating System Prin.*, Pacific Grove, CA, 1979.

[Lis81] Liskov, B., On Linguistic Support for Distributed Programs, *IEEE Trans. on Software Eng. SE-8*, 3 (July 1981).

[Lom79] Lomet, D. B., Coping with Deadlock in Distributed Systems, in *Database Architecture*, Bracchi, G. and Nijjsen, G. M. (ed.), North-Holland, 1979.

[MaI80] Marsland, T. A. and Isloor, S., Detection of Deadlocks in Distributed Database Systems, *INFOR 18*, 1 (Feb. 1980).

[McD81] McDermid, J. A., Checkpointing and error Recovery in Distributed Systems, *Proc. 2nd Int. Conf. on Distributed Systems*, 1981.

[MeM79] Menasce, D. and Muntz, R., Locking and Deadlock Detection in Distributed Databases, *IEEE Trans. on Software Eng. SE-5*, 2 (May 1979), 195-202.

[Mos81] Moss, J. E. B., Nested Transactions: An Approach to Reliable Distributed Computing, MIT/LCS/Tech. Rep.-26, MIT Lab. for Computer Science, Cambridge, Mass., April 1981 .

[Mun79] Munz, R., Gross Architecture of the Distributed System VDN , in *Database Architecture*, Bracchi, G. and Nijjsen, G. M. (ed.), North Holland, 1979.

[Ran75] Randell, B., System Structure for Software Fault Tolerance, *IEEE Trans. on Software Eng. SE-1*, 3 (June 1975), 220-232.

[RLT78] Randell, B., Lee, P. A. and Treleaven, P. C., Reliability Issues in Computing Systems Design, *Computing Surveys 10*, 2 (June 1978), 123-165.

[ReS81] Reed, D. and Svobodova, L., Swallow: A Distributed Data Storage System for a Local Network, in *Local Networks for Computer Communications*, West, A. and Janson, P. (ed.), North Holland, 1981, 355-373.

[Ree83] Reed, D., Implementing Atomic Actions on Decentralized Data, *ACM Trans. Computer Systems 1*, 1 (Feb. 1983), .

[Reu81] Reuter, A., Recovery Architecture for Database Systems, *Proc 6th ACM European Regional Conf. on Systems Architecture*, 1981.

[RiS77] Ries, D. and Stonebraker, M., Effects of Locking Granularity in Database System, *ACM Trans. Database Systems 2*, 3 (Sept. 1977), 233-246.

[RiS79] Ries, D. and Stonebraker, M., Locking Granularity Revisited, *ACM Trans. Database Systems 4*, 2 (June 1979), 210-227.

[RiS82] Ries, D. R. and Smith, G. C., Nested Transactions in Distributed Systems, *IEEE Trans. on Software Eng. SE-8*, 3 (July 1982).

[RSL78] Rosenkratz, D. J., Sterns, R. E. and Lewis, P. M., System level Concurrency Control for Distributed Database Systems , *ACM Trans. Database Systems 3*, 2 (June 1978), 178-198.

[RBF80] Rothnie, J. B., Bernstein, P. A., Fox, S., Goodman, N. and Hammer, M., Introduction to a System for Distributed Databases (SDD-1), *ACM Trans. Database Systems 5*, 1 (March 1980), 1-17.

[Sch78] Schlageter, G., Process Synchronization in Database Systems, *ACM Trans. Database Systems 3*, (1978).

[Sch79] Schlageter, G., Enhancement of Concurrency in Database Systems by the Use of Special Rollback methods, in *Database Architecture*, Bracchi, G. and Nijjsen, G. M. (ed.), North Holland, 1979.

[Sel80] Selinger, P., Replicated Data , in *Distributed Databases*, Draffan, I. W. and Poole, F. (ed.), Cambridge University Press, 1980.

[Shr81] Shrivastava, S. K., Structuring Distributed Systems for Recoverability and Crash Resistance, *IEEE Trans. on Software Eng. SE-7*, 4 (July 1981), 436-447.

[ShP82] Shrivastava, S. K. and Panzieri, F., The Design of a Reliable Remote Procedure Call Mechanism, *IEEE*

Trans. on Computers C-31, 7 (July 1982), 692-697.

[SkS81] Skeen, D. and Stonebraker, M., A Formal Model of Crash Recovery in Distributed Systems, Memo UCB/ERL M80/48, Electronics Research Lab., Univ. California, Berkeley, 1981.

[Ske81] Skeen, D., Non-blocking Commit Protocols , *ACM SIGMOD Int. Conf. on Management of Data* , 1981, 133-141.

[SmB81] Smith, J. A. and Bernstein, P. A., Multibase: Integrating Heterogeneous Distributed Database Systems, *Proc. AFIPS Conf.*, 1981.

[Spa80] Spaccapietra, S., Heterogeneous Database Distribution, in *Distributed Databases*, Draffan, I. W. and Poole, F. (ed.), Cambridge University Press, Cambridge, England, 1980.

[SpS83] Spector, A. and Schwarz, P., Transactions: A Construct for Reliable Distributed Computing, *SIGOPS Review 17*, 2 (April 1983), .

[Sto79] Stonebraker, M., Concurrency Control and Consistency of Multiple Copies of Distributed INGRES , *IEEE Trans. on Software Eng. SE-5* , 3 (May 1979).

[Sto80] Stonebraker, M., Homogeneous Distributed Database Systems, in *Distributed Databases*, Draffan, I. W. and Poole, F. (ed.), Cambridge University Press, Cambridge, England, 1980.

[Tho79] Thomas, R. H., A Majority Consensus approach to Concurrency Control for Multiple copy Databases, *ACM Trans. Database Systems 4*, 2 (June 1979), 180-209.

[ToS81] Toan, N. G. and Sergeant, G., Distributed Architecture and Decentralized Control for a Local Network Database System, *Proc. ACM European Conf. on Systems Architecture*, 1981.

[TGG82] Traiger, I. L., Gray, J., Galtieri, C. A. and Lindsay, B. G., Transactions and Consistency in Distributed Database Systems, *ACM Trans. Database Systems 7*, 3 (Sept 1982), 323-342.

[TsB82] Tsai, W. and Belford, G. G., Detecting Deadlocks in a Distributed System, *Proc. of IEEE INFOCOM*, Los Vegas, Nevada, April 1982.

[TKN81] Tsuruka, K., Kaneko, A. and Nishihara, Y., Dynamic

Recovery Schemes for Distributed Processes, *IEEE Symp. on Reliability in Distributed Software and Database Systems*, Pittsburgh, Penn., June 1981.

[Ull82] Ullman, J. D., in *Priciples of Database Systems*, 1982, Computer Science Press.

[Ver77] Verhofstad, J. S. M., Recovery and Crash Resistance in a Filing System, *ACM SIGMOD Int. Conf. on Management of Data*, Toronto, Canada, 1977.

[Ver78] Verhofstad, J. S. M., Recovery Techniques for Database Systems, *Computing Surveys 10*, 2 (June 1978), 167-195.

[Ver79] Verhofstad, J. S. M., Recovery Based on Types, in *Database Architecture*, Bracchi, G. and Nijjsen, G. M. (ed.), North Holland, 1979.

[Vin81] Vines, D. H., A Dataflow Solution to Implementing Distributed Queries , *Proc. 6th Berkeley workshop on Distributed Data Management and Computer Networks* , Berkeley, Calif. , 1981.

[Wat81] Watson, R. W., Distributed System Architecture Model, in *Distributed Systems: Architecture and Implementation*, Springer Verlag, 1981, 10-43.

Appendix

Symbols and Notations

$\exists$ ..... there exists

$\wedge$ ..... logical and

$\forall$ ..... for all

$\times$ ..... cross product

ϵ ..... element of

$|\ |$ ..... Cardinality of a set

CT ..... Compensating Transaction

TM ..... Transaction Manager

DM ..... Data Manager

BM ..... Buffer Manager

DBMS ..... Data Base Management System

DDBMS ..... Distributed DBMS

DDB ..... Distributed Data Base